NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

PC104 CONTROL ENVIRONMENT DEVELOPMENT AND USE FOR TESTING THE DYNAMIC ACCURACY OF THE MICROSTRAIN 3DM-GX1 SENSOR

by

Jonathan Shaver

June 2007

Thesis Advisor:	Xiaoping Yun
Co-Advisors:	Matthew Feemster
	Douglas Fouts

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE PC104 Control Environment Development and Use for Testing the Dynamic Accuracy of the MicroStrain 3DM-GX1 Sensor	**5. FUNDING NUMBERS**
6. AUTHOR(S) Jonathan Shaver	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

There is a need for a standard, accurate test bench for inertia-based orientation sensors. Static accuracy testing of these sensors is straightforward but dynamic accuracy testing is more difficult. A test bench system is developed with encoders and a PC104 computer under the QNX Neutrino real-time operating system. A MicroStrain 3DM-GX1 inertial sensor was used as the sensor to be tested. The dynamic error of this sensor was accurately recorded and found to be a function of the sensor velocity and acceleration.

14. SUBJECT TERMS PC104, MicroStrain, 3DM-GX1, 16-Bit absolute encoder, Controls environment, inertia-based orientation sensor	**15. NUMBER OF PAGES** 133
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

PC104 CONTROL ENVIRONMENT DEVELOPMENT AND USE FOR TESTING THE DYNAMIC ACCURACY OF THE MICROSTRAIN 3DM-GX1 SENSOR

Jonathan A. Shaver
Ensign, United States Navy
B.S., United States Naval Academy, 2006

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
JUNE 2007

Author: Jonathan Shaver

Approved by: Xiaoping Yun
Thesis Advisor

Matthew Feemster
Co-Advisor

Douglas Fouts
Co-Advisor

Jeffery B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

There is a need for a standard, accurate test bench for inertia-based orientation sensors. Static accuracy testing of these sensors is straightforward but dynamic accuracy testing is more difficult. A test bench system is developed with encoders and a PC104 computer under the QNX Neutrino real-time operating system. A MicroStrain 3DM-GX1 inertial sensor was used as the sensor to be tested. The dynamic error of this sensor was accurately recorded and found to be a function of the sensor velocity and acceleration.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my wife, Jillian Shaver, for helping me through this project in every non-technical way possible and always being there for me when I needed her. I would not have had the strong drive and motivation to work hard without her little pushes to keep me on track.

I would also like to thank Dr. Yun and James Calusdian for being there to answer all my questions and helping me in a great way with all the problems I encountered throughout my thesis. Dr. Yun has been a great teacher and mentor during my time at the Naval Postgraduate School. James was always there to help with any technical problems I encountered. He was quick to drop his own work to help me with problems and he thought nothing of working outside of normal working hours to build various pieces required for this project setup.

Dr. Feemster helped push me in the direction of attending the Naval Postgraduate School while I was attending the United States Naval Academy. With his help and dedication I was able to receive an invitation to attend the Naval Postgraduate School, for that I would like to thank him. He was also the starting motivation for this thesis idea.

I would like to thank Dr. Fouts for being a co-advisor on this project. I would finally like to thank my family, Grandma and Grandpa Shaver, Grandpa Jackson and Shirley Russell, "Granny," for their unending support in my journey to where I am today. It would not have been possible without them.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARRY

The purpose of this project was to develop a control systems environment complete with encoder capabilities on a PC104. This system's capabilities were tested by using it to measure the dynamic accuracy of the MicroStrain 3DM-GX1 orientation sensor against the benchmark of an encoder. The digital input/output and the digital-to-analog conversion circuits and the RS-232 port of the PC104 were used to accomplish this task.

In this project, a Diamond Systems Prometheus PC104 with a data acquisition circuit was used as a target machine. The main encoder used was a 16-bit absolute encoder by Gurley Precision Instruments. The encoder communication and control was performed with the digital input/output circuit. The MicroStrain 3DM-GX1 sensor communications all occurred through the RS-232 port. The MicroStrain sensor was attached to the end of a wooden pendulum and the encoder served as the pivot point of this pendulum. This allowed the angles of the two sensors to move synchronously with one another. The pendulum was set in motion and the position of the two sensors were recorded and output to a MATLAB compatible file.

The results yielded an error of the MicroStrain sensor well within the specifications established by the company. The worst error was 1.3019° when the pendulum was at a length of one foot. It was discovered that the error is a function of the velocity, acceleration, and flexure of the pendulum.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

The motivation for this thesis originates from two different driving sources, the need for a PC104 controls environment at the United States Naval Academy and the need for a standard, accurate test bench for inertia-based orientation sensors.

Currently, the United States Naval Academy Systems Engineering Department uses the Rabbit 2000 microprocessor as a control systems platform. This system has many limitations that create the need for a newer, more powerful system. The first major limiting factor of the embedded systems the school currently has is the limit of control routine execution rate. The systems currently used are not able to execute control routines any faster than approximately 1 kHz. This speed is fast enough for most routines used but there are many new routines that could be executed if the control routine could execute at a faster rate. Other limiting features include difficulty interfacing with MATLAB, communication speed, portability, and minimal data storage. A PC104 with a data acquisition circuit would be the solution. For this system to be incorporated into that classroom at the Naval Academy, the complexities of the inter-circuit and inter-system programming must first be reduced into simple user-friendly function calls. A class of functions that operate the digital to analog converters, analog to digital converters, digital input/output, and encoder input are necessary.

Without this type of function library the students would need a much more extensive knowledge base in computer programming and operation. A complied function library will allow users with basic computer programming knowledge to program a complex control routine on the embedded system.

The second source of motivation is more of an industry wide source. There is a need for a standard, accurate test bench for inertial sensor units, as Yun and Bachmann identified in Ref. 1. The MARG project at the Naval Postgraduate School shows a 9° error between the inertial sensor and the test bench.[1] The paper in reference 1 indicates the error is greatest during the dynamic motion periods. One of the problems is the test bench does not give output of the angle. The test apparatus can only be set to turn to specific angles. The intermittent position information is not available. This means the position of the test bench must be modeled. In the case of reference 1, the test bench motion is modeled as linear motion. This is not a correct model because the test bench must accelerate and decelerate. The lack of this information could be the cause of the error discovered. Creating a test bench that is capable of recording or outputting intermittent position information would eliminate this problem and the error from this problem would be reduced, allowing the analysis of the sensor to be more accurate.

The MicroStrain 3DM-GX1 sensor is an inertial-based orientation sensor that could be used in the project of reference 1. This sensor has been used in human motion tracking in other projects[2]. To model and track the information more accurately the error of the sensor would have to be known. The MicroStrain company does not distribute any information about the sensor operation with respect to the dynamic motion, besides the dynamic accuracy. It is not known exactly how accurate the sensor is during motion. Static accuracy can be easily determined by keeping the sensor still and reviewing the output data. Dynamic accuracy is harder to measure because the sensor position has to be measured while in motion. An accurate test bench system must be created that will allow the position measurement and the sensor readings to be correlated as closely as possible. An encoder in conjunction with a fast and powerful control system would allow this analysis.

B. GOALS

The main goals of the project are:

- Develop a user-friendly controls environment on a PC104, complete with data input and output.
- Develop and create a test bench for inertia-based orientation sensor.
- Develop an interrupt service routine that will service the serial universal asynchronous receiver transmitter, compatible with the RS-232 standard.
- Test the dynamic accuracy of the MicroStrain 3DM-GX1 orientation sensor.

C. CHAPTER DESCRIPTIONS

Chapter I is the introduction and motivation for this project.

Chapter II discusses most platform specific information. It gives a background and description of the PC104 used in this project. It also describes the QNX

Neutrino operating system and a short introduction to what a real-time operating system is and why this project requires such an operating system.

Chapter III describes in detail all the apparatus used or developed in the project. It discusses the host machine, the target PC104 input, output, and boot indication program, the MicroStrain 3DM-GX1 sensor, the encoders used in the project and finally the pendulum constructed.

Chapter IV describes all the test related information. This chapter is where the actual setup of the apparatus is discussed. A complete description of the data acquisition program and the attached interrupt service routine is included. The chapter finishes with plots and explanation of all the results.

The final chapter, Chapter V, contains the conclusions and the recommendations for future work.

Appendix A is a complete version of the 16-Bit encoder program used to record the MicroStrain 3DM-GX1 sensor dynamic error. The entire code was written in C++.

Appendix B is a complete version of the ***control.wse*** file which contains many of the functions created and used in this project.

II. PLATFORM INTRODUCTION

A. THE PC104

1. Introduction

The heart of this project is a PC104. It is the system that controls and records all information needed to make observations and calculations. The PC104 interacts with the apparatus to be tested through digital input/output (DIO) ports, analog to digital converters (ADC), and digital to analog converters (DAC).

2. Background

The PC104 technology was first created in 1987 by Ampro Computers.[3] This company was founded in 1983 for the purpose of manufacturing compact single board computers to be used in embedded systems. The MiniModule was the first PC compatible embedded system with the capability of modular expansion. For the purpose of this thesis PC refers to Personal Computer, a term and standard developed by the IBM company. The form factor of the MiniModule is one of the features that classified it as the first PC104. The actual board measured 3.6 inches by 3.8 inches, which is now the PC104 form factor.[3]

Even though the first PC104 was produced in 1987 it was not a common system in the embedded control industry until the PC104 Consortium was founded by 12 companies in February 1992.[4] A month later, in March 1992, the PC104 Consortium published the "PC104 Specifications" which established and launched the vast popularity of this new standard.[4]

The PC104 gets its name from some of its more distinguishing specifications. The system is compatible with any computer compatible with the PC standard introduced by IBM. The second part of the name is derived from the pin specification for inter-module communication. There is a set configuration of 104 pins which enables communication between the CPU module and any other module.

3. Prometheus PC104

The PC104 used in this project is a Diamond Systems Prometheus PC104. It was selected because it has a powerful data acquisition board packed with it. The operating system is the QNX Neutrino 6.2.1 real-time OS. It has an internal 6 GB hard drive connected to the IDE port. The hard drive has a separate power supply and is not housed within the vendor supplied PC104 casing. The hard drive is too large to fit inside. The OS was loaded onto the hard drive while it was connected to anther desktop PC. This method of loading the OS was easier than connecting a CD-ROM directly to the PC104. Once the OS was loaded, the hard drive was connected to the PC104.

The specifications of the PC104 were taken directly from the Prometheus CPU user manual in Ref. 5:

Processor Section

- ♦ 486-DX2 processor running at 100MHz with co-processor
- ♦ Pentium class platform including burst-mode SDRAM and PCI-based IDE controller and USB
- ♦ 32MB SDRAM system memory

- ♦ 50MHz memory bus for improved performance
- ♦ 2MB 16-bit wide integrated flash memory for BIOS and user programs
- ♦ 8KB unified level 1 cache

I/O

- ♦ 4 serial ports, 115.2kbaud max
- ♦ 2 ports 16550-compatible, 2 ports 16850-compatible with 128-byte FIFOs
- ♦ 2 full-featured powered USB ports
- ♦ 1 ECP-compatible parallel port
- ♦ Floppy drive connector
- ♦ IDE drive connector (44-pin version for notebook drives)
- ♦ Accepts solid-state flashdisk modules directly on board
- ♦ 100BaseT full-duplex PCI bus mastering Ethernet (100Mbps)
- ♦ IrDA port (requires external transceiver)
- ♦ PS/2 keyboard and mouse ports
- ♦ Speaker, LEDs

System Features

- ♦ Plug and play BIOS with IDE autodetection, 32-bit IDE access, and LBA support
- ♦ Built-in fail-safe boot ROM for system recovery in case of BIOS corruption
- ♦ User-selectable COM2 terminal mode
- ♦ On-board lithium backup battery for real-time-clock and CMOS RAM
- ♦ ATX power switching capability
- ♦ Programmable watchdog timer
- ♦ Power surge monitor for fail-safe operation
- ♦ Zero wait-state capability for flash memory and PC/104 bus
- ♦ +5V-only operation

- ♦ Extended temperature range operation (-40 to +85oC)
- ♦ Cable-free operation when used with Diamond Systems' PNL-Z32 Panel I/O board

The PC104 also has a data acquisition circuit for I/O operations. This circuitry communicates with the PC104 through the ISA bus. The circuit diagram of the Prometheus, which depicts this circuit, is seen in Figure 1. More details of the data acquisition circuit are discussed in a later chapter.

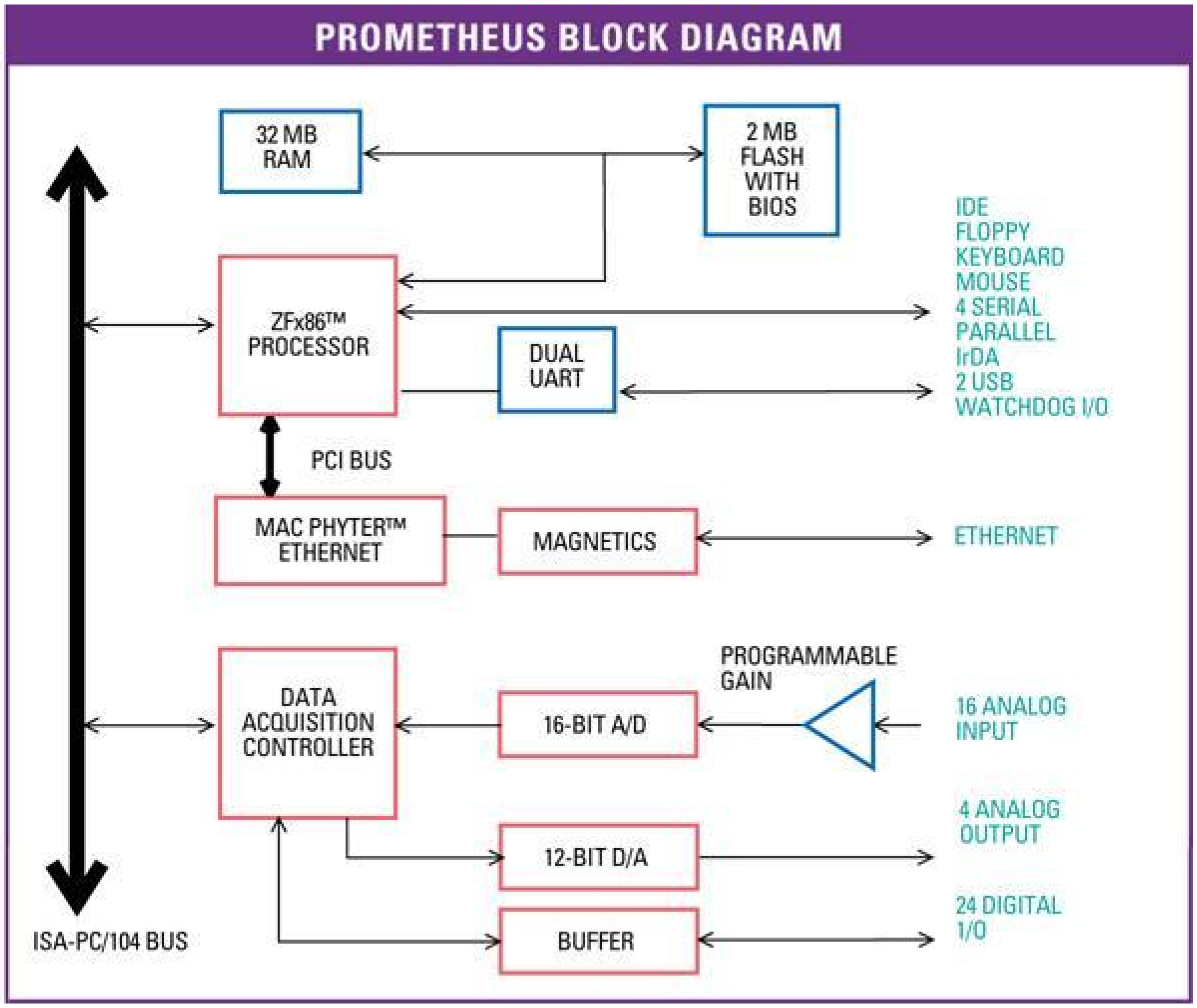


Figure 1. Prometheus PC104 Block Diagram [From Ref. 5].

A photo of the actual PC104 used in this project is seen in Figure 2.

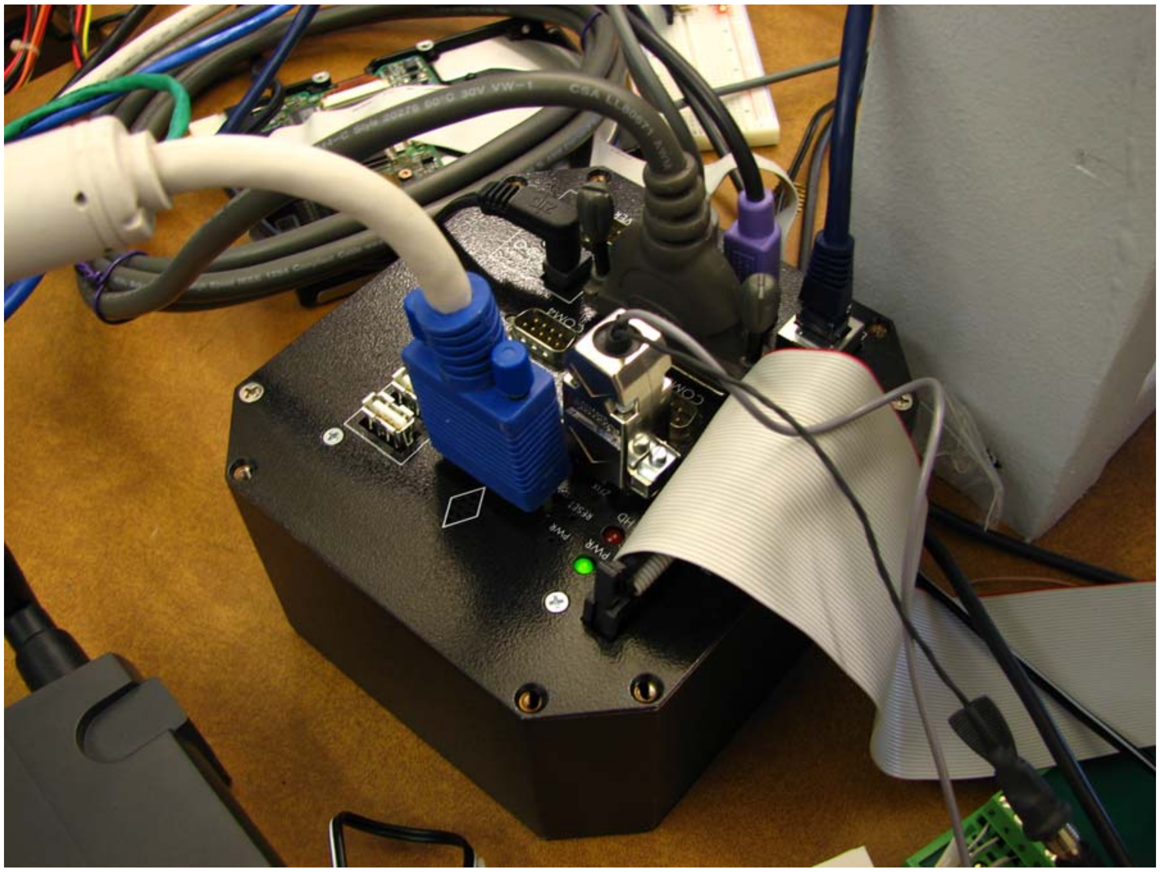


Figure 2. Actual PC104 Used in the Project.

B. THE QNX OPERATING SYSTEM

1. Introduction

The operating system (OS) that was used for this project was QNX Neutrino Real-Time Operating System (RTOS) version 6.2.1A. It is bundled with the QNX Momentics Professional Edition Integrated Development Environment (IDE). This operating system was chosen because it is a real-time OS and it was free through an education grant given by the QNX Software Systems Company.

2. QNX Neutrino RTOS v.6.2.1A

The QNX Neutrino v 6.2 was developed and released in 2002. QNX was founded in 1980 and since then has been the leader in the industry of real-time microkernel operating systems.[6] The real-time microkernel aspect of the OS is vitally important to the application described in this thesis. A deterministic execution of code under an RTOS allows the recording and syncing of the measurements between two different I/O devices to be as close as possible.

One of the main advantages of an RTOS is its predictability. An RTOS is designed to guarantee the execution of a computation in a reasonable amount of time if there are no external influences, such as interrupts. Since external influences such as interrupts can be disabled, the programmer has the ability to ensure a deterministic execution environment by implementing the proper setting. This may be disabling all interrupts or at least controlling necessary interrupts. This extent of control ensures that any delays in a system are not caused by the OS.[7]

The deterministic timing and execution of instructions relies partly on the scheduling scheme of the OS. A prioritized system of scheduling must be utilized. Higher priority items must be allowed to execute before lower priority items. This is a necessity for an RTOS. This gives the OS a way to ensure that all real-time sensitive instructions are actually executed deterministically. A real-time thread or process would be assigned a higher priority to ensure that it is executed when it needs to. The only thing that would be able to potentially interrupt this execution would be a system call.[7]

Priority inversions must also be controlled in an RTOS. This is where a high priority thread is trying to use a resource that is already allocated to a low priority thread and a medium priority thread is executing. In this case, the low priority thread would not be allowed to complete and free the resources until the medium thread resource is completed first. This would cause many problems with the system and cause the system to lose its real-time characteristic. High priority threads are usually threads designed to run real-time. If the thread has to wait for lower priority threads to complete first, timing of the high priority thread would no longer be guaranteed. To ensure a system is real-time, priority inversion must be controlled in such a way that a real-time thread will never have to wait for a lower priority thread. This is either accomplished by the lower priority thread giving up the resource or the lower priority thread taking on the higher priority's priority through priority inheritance.[7]

Another requirement for a real-time operating system is for the system to have a periodic division of processor time that can be allocated to processes. A portion of this time must be reserved for real-time processes. It must also ensure that non-schedulable instructions, such as microkernel calls, must not violate this time limit. When any instruction executes, whether it is in a kernel call, a real-time activity, or non-real-time activity, if it takes more than the predetermined period of time it starts to affect the characteristics of a real-time system. No longer are all execution times guaranteed because real-time processes are not able to execute when expected. This is true even for other real-time processes. If there are two real-time processes running at the same time, they must equally share the time on the processor. If they do not, one of the processes will start to show non-real-time characteristics because the execution will not start when expected. Under an RTOS, programs are designed to run on a strict schedule because the time between executions is critical for some particular reason.[7]

Steve Furr develops five critical system requirements for an RTOS in his paper ***What is Real Time and Why Do I Need It? from Ref. 7***:

1 The OS must support fixed-priority preemptive scheduling for tasks (both threads and processes, as applicable).

2 The OS must provide priority inheritance or priority-ceiling emulation for synchronization primitives.

3 The OS kernel must be pre emptible.

4 Interrupts must have a fixed upper bound on latency.

  a. By extension, support for nested interrupts is required.

5 Operating system services must execute at apriority determined by the client of the service.

  a. All services on which the client is dependent must inherit that priority.

  b. Priority inversion avoidance must be applied to all shared resources used by the service.

Requirement 1 indicates that scheduling must be fixed priority and preemptive. A scheduling priority is definitely needed, as discussed earlier. The fixed priority is a more well-defined requirement. It is required to be fixed so that there is a set standard and structure for the programmer to follow when designing a real-time program and application. This standard fixed priority allows the programmer to run the program at a known priority, allowing real-time execution. The second half of this requirement is also very important. If a lower priority process is taking too long, or a higher priority process must execute, then there must be a way for immediate execution. This is accomplished with the preemptive nature of the scheduling routine. Preemptive scheduling allows the operating system to halt a currently running process or thread and execute a higher priority process or thread, after which it restores the state of the halted process or thread and it is able to continue execution. This is important because it is another characteristic of the OS that ensures the timing guarantee of an RTOS.

Requirement 2 was discussed earlier in the paper. Essentially, there must be priority inheritance to avoid a priority inversion condition.

The third requirement further extends the bounds of the first requirement. The kernel is a type of layer that allows interaction between system critical hardware and the programs. It controls communication between programs and hardware such as the CPU, memory, and other devices. Since is it conceivable that many programs will have some type of interaction with one of these system critical hardware devices, it makes sense that the kernel must have preemption capabilities. If a real-time process relies on a kernel call for proper real-time execution, the kernel must be able to preempt a currently running process or thread. This helps to guarantee discrete execution of the real-time process or thread.

The fourth requirement is needed to limit the invasion of interrupts. In real-time operating systems, interrupts are very important. In many applications, interrupts are used to guarantee the operation is performed at a real-time pace. This will be the case in this project. The majority of the program is run by interrupts. If the interrupts are processed instantly and are not very complicated, the system will operate at a predictable, deterministic rate. For interrupts to occur in a real-time OS and not affect the real-time characteristic, the interrupt handler must respond quickly. This is the reason to place an upper limit on the latency.

The final requirement extends the earlier discussion about priorities of the OS services. Under a real-time OS, all processes, thread, and operations must execute in a controlled, deterministic manner.

The QNX Neutrino OS has all of the characteristics. This means if a program is created and run properly, it will run in real-time. This is crucial for this project because the exact time and sensor positions must be recorded. If there is any delay in the program the time and the position may not correlate. In this project there are two sensors being read and correlated with time and position, so it is even more vital that the program runs in real-time.

The entire program will be written in C++ with the QNX Momentics IDE. When the programs are compiled they will execute on the target PC104 system that will be in constant communication with the host Momentics instance through the Ethernet connection. Specific aspects of the communication, the program code, and execution will be discussed in later chapters.

THIS PAGE INTENTIONALLY LEFT BLANK

III. THE APPARATUS

A. HOST MACHINE

The QNX programming is all performed on a host machine. This is necessary because the target machine, the PC104, does not have enough memory to run the QNX Momentics Integrated Development Environment (IDE). The PC104 only has 32MBs of RAM and the IDE requires far more than that. QNX recommends 256MB of RAM. The PC104 uses 19MB of the RAM just for the QNX operating system to run. When the IDE is launched on the PC104, the memory is maxed out and the IDE will quit loading during its startup phase of loading. The memory is then freed of the partially loaded program. The target's processor is only 100MHz which would also limit the productivity of the system. While processor speed will not necessarily stop the program from loading or working correctly, the program will most likely run so slowly that productivity would be severely limited.

The host machine is a Dell Dimension 8100 with an Intel Pentium 4, 1.70 GHz processor and 2GB of RAM. It uses a 3-Com 100 Base-T Ethernet Card for host to target communication. The only input devices used are a mouse and keyboard and the only output device used is the VGA output of the on-board graphics card. This is all exclusive of the I/O Ethernet card. This Dell system has an on-board Ethernet card that has been disabled in the BIOS to decrease the likelihood of hardware conflicts.

B. TARGET MACHINE – PROMETHEUS PC104

The target machine is the PC104 described a previous chapter. This target machine is the location where all the developed code is actually run. The QNX Momentics IDE can be set up to communicate between the host and target directly through the Ethernet connection.

1. Unidirectional Input Devices

While the PC104 is capable of using a mouse and keyboard as input this particular setup does not include either. The purpose of this project is to have a PC104 that is entirely remote operable. A mouse and keyboard would serve no purpose in a remote application as the user would not have access to them. The PC104 does not use the USB ports because all peripherals are accessed through other ports.

2. Unidirectional Output Devices

The PC104 does have a VGA adapter card but this card was not used for any step other than the initial set up of the PC104. There are some applications where a VGA monitor may be useful but because this PC104 setup is designed to be operated in a remote environment, most likely the user will not be able to be in the same location of the monitor or the system will not be able to support the monitor. Many remote applications of this PC104 will not be able to support a monitor because of the power requirements. Many of the applications of this PC104 require the lowest possible power draw of the system. Monitors will increase system power requirements while not adding much benefit to the system.

If a user is in the same location as the PC104, the user will be able to remotely log in to the system for diagnostics. When executing code from the host machine, all console output is directed to the host machine erasing the need for a monitor as a console out for program execution.

There is one instance when a monitor would be very useful, that is the boot phase of the PC104 startup. Since the PC104 only has a 100 MHz processor, it takes much more time to boot to the user screen than a modern day personal computer does. Without a monitor it is very difficult to know exactly when the machine has finished booting up. A monitor would obviously be one solution to this problem. Since this project aims to decrease all power consumption a new boot completion technique must be used. The solution was to develop an LED indication of when the system has finished booting.

a. Boot Completion Indicator

The boot completion indicator output is driven by the parallel port. For this PC104 the parallel port is set at location 0x378. The indicator is a set of eight red LEDs that are driven from the power supplied by the port. Each LED is driven by one of the eight data output ports of the parallel port. On a standard 25-pin parallel port the data output is seen on pins 2-8. Ground can be connected to any pin between pins 18-25. This project uses pin 18 for ground. The circuit includes a resistor between the LED and the positive end of the supply, as seen in Figure 3. The resistor is there to limit the amount of current that flows through the LED.

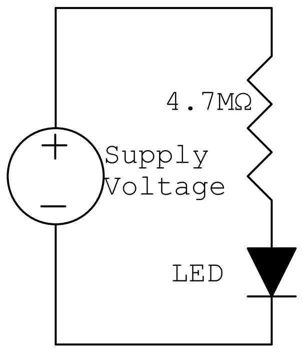


Figure 3. LED Circuit for boot completion indicator.

LEDs have very low internal resistance and without a resistor an LED could draw enough current to either damage itself or damage the parallel port hardware. An array of eight LEDs that are all individually controlled by a separate output line of the parallel port was the method of choice because this helps to decrease false indication of the system state. If only one LED was used there would be a higher risk involved that either the LED burned out or that particular data line is faulty. Including all eight available lines and eight LEDs lowers the risk that the output, either an LED on or off, of the overall system is faulty. A completely faulty indication, in the one LED system or the eight LED system, will decrease system reliability because the user will not know for sure whether the system is fully booted or not.

The program that was used to operate this circuitry is listed below:

```
#include <unistd.h> //for delay()
#include <hw/inout.h> //for out8()
#include <sys/neutrino.h> //for ThreadCTL()

```

```
5:  int main()
6:  {
7:    //GAIN ROOT PERMISSION FOR I/O CONTROL
8:    ThreadCTL(_NTO_TCTL_IO,NULL);
9:    for(int i=0;i<10;i++) //FOR 10 ITTERATIONS
10:   {
11:        out8(0x378,0); //All LEDS OFF
12:        delay(100); //DELAY 100ms
13:        out8(0x378,255); //ALL LEDS ON
14:        delay(100); //DELAY 100ms
15:   }
16:   return 1;
17: }
```

When the program starts it needs to get root permission to perform I/O operations. This is accomplished y the call to *ThreadCTL()* at line 8. This function is specific to the QNX Neutrino operating system. Once the I/O permission is obtained the program continues. The *for* loop started at line 9 gives the indicator the functionality to signal exactly when the OS is fully loaded by blinking ten times. This task is accomplished by the blinking cycle contained within the *for* loop. The call to *out8(0x378,XXX)* at lines 11 and 13 send an 8 bit output byte to the parallel port found at location 0x378. The second parameter of the function is the value of the 8-bit word that is sent. Zero is binary 00000000, meaning all the LEDs will be off because all outputs are low (0). Line 13 sends 255 which is 11111111, turning all the LEDs in the array on because all bits of the parallel port data output are high (1). When the output is high it outputs 5V, enough to turn the LEDs on. When the program completes execution, all LEDs are left on because the last data packet sent to the parallel port was 255. This constant burning LED array indicates that the PC104 is fully booted and the user may start executing other programs. The array in this state can be found in Figure 4.

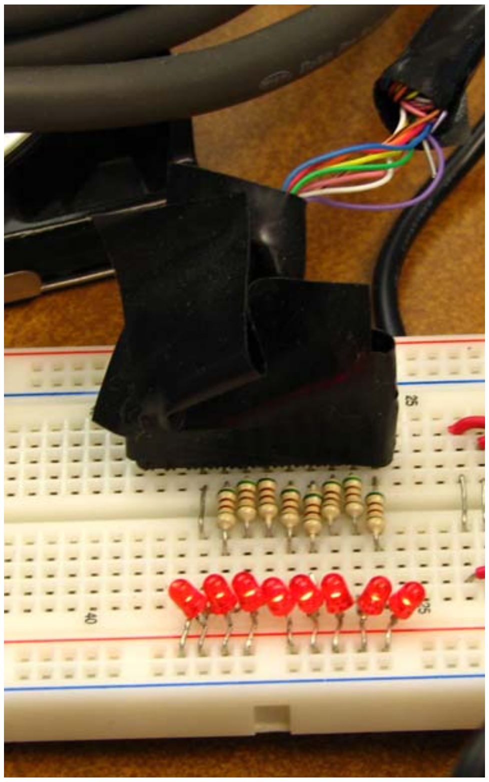


Figure 4. Boot Completion Indicator.

The LEDs will remain on as long as the user does not modify the port within a program. The operating system's interaction with the parallel port is stopped just before the start up indicator program is started. The OS startup script is located at /ect/rc.d/rc.sysinit. To ensure proper operation of the boot indication program the line ***slay devc-par*** must first be included in ***rc.sysinit.*** This disconnects the OS handle to the parallel port. If this process is not killed, the OS may interfere with the proper operation of the program, or the lasting effect of continuing the LEDs in the on state. The next line that must be added to ***rc.sysinit*** is ***startind***. This line tells the OS to start the boot completion indicator program execution. This is how the proper startup of the OS can be indicated to the user without the use of a monitor.

3. Bidirectional Input/Output Devices

There are four main devices on board the PC104 that are bidirectional I/O devices. They are the serial communications device, data acquisition circuit, Ethernet communications device, and the universal serial bus (USB) device. The USB ports were not setup or used because they serve no immediate benefit to the project.

a. Ethernet Port and IP Address

The Ethernet port was established on the target with a static IP of 192.168.0.2. Creating a static IP address instead of a dynamic IP address allows the system to mesh more easily. If the IP of the target was not static, the user would have to find a way to retrieve the IP address. With no keyboard or monitor in the system it would be difficult to retrieve the IP address if it were dynamically assigned by an outside device. Therefore, assigning a static IP address is more reliable.

b. RS-232 Serial Port

The PC104 system has four RS-232 serial ports on the I/O module. Serial port 1 was used for the communications between the PC104 and the MicroStrain sensor. The other ports were not needed. The serial port 1 (COM1) is built into the ZF Micro CPU chip on the PC104 module. It is a 16550 standard serial universal asynchronous receiver/transmitter (UART) with a 16-byte first-in-first-out buffer.[5] Actual communications of this device are discussed in a later chapter.

c. Data Acquisition Circuit

The data acquisition circuit is part of the Prometheus PC104 package. This is the device that performs all of the digital input/output. This circuit is also capable of analog to digital conversions and digital to analog conversions. The complete list of specifications are seen below. This list was taken from the Prometheus User Manual, Ref. 5.

Analog Inputs	
No. of inputs	8 differential or 16 single-ended (user selectable)
A/D resolution	16 bits (1/65,536 of full scale)
Input ranges	Bipolar: ±10V, ±5V, ±2.5V, ±1.25V Unipolar: 0-10V, 0-5V, 0-2.5V
Input bias current	50nA max
Maximum input voltage	±10V for linear operation
Overvoltage protection	±35V on any analog input without damage
Nonlinearity	±3LSB, no missing codes
Drift	5PPM/oC typical
Conversion rate	100,000 samples per second max
Conversion trigger	software trigger, internal pacer clock, or external TTL signal
FIFO	48 samples; programmable interrupt threshold
Analog Outputs	
No. of outputs	4
D/A resolution	12 bits (1/4096 of full scale)
Output ranges	Unipolar: 0-10V or user-programmable Bipolar: ±10V or user-programmable
Output current	±5mA max per channel
Settling time	4µS max to ±1/2 LSB

Relative accuracy	±1 LSB
Nonlinearity	±1 LSB, monotonic
Digital I/O	
No. of lines	24
Compatibility	3.3V and 5V logic compatible
Input voltage	Logic 0: -0.5V min, 0.8V max; Logic 1: 2.0V min, 5.5V max
Input current	±1μA max
Output voltage	Logic 0: 0.0V min, 0.4V max; Logic 1: 2.4V min, 3.3V max
Output current	Logic 0: 12mA max; Logic 1: -8mA max
I/O capacitance	10pF max
Counter/Timers	
A/D pacer clock	24-bit down counter
Pacer clock source	10MHz, 1MHz, or external signal
General purpose	16-bit down counter
GP clock source	10MHz, 100KHz, or external signal
General	
Power supply	+5VDC ±5%
Current consumption	0.7A – 1.1A typical
Operating temperature	-40 to +85oC
Operating humidity	5% to 95% noncondensing

A block diagram of the data acquisition circuit can be found in Figure 5.

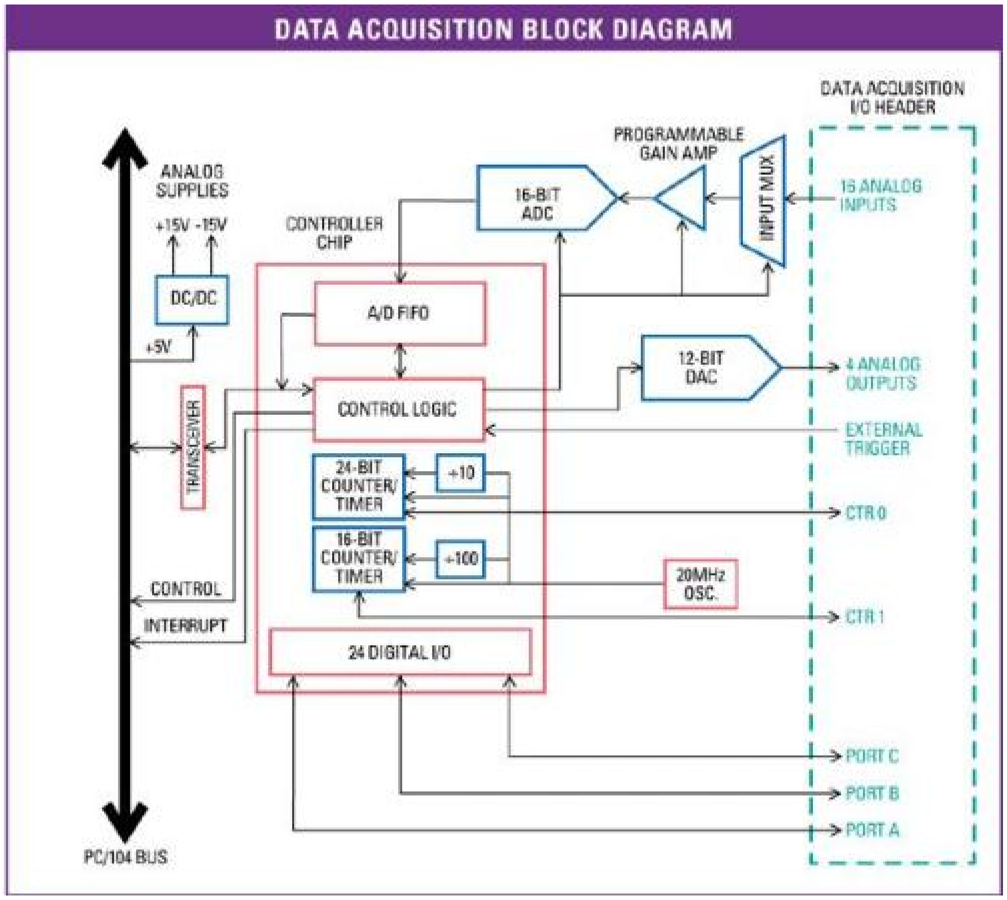


Figure 5. Data Acquisition Block Diagram [From Ref. 5].

The data acquisition circuit communicates with the CPU over the PC104 bus. All control and communication to the circuit is performed through register read and writes, as the I/O is mapped to registers. All of the registers are an offset of the base address of the circuit. In the case of this project the base address is 0x280. I/O time is more dependent on the hardware of the circuit than the connection between the circuit and the CPU. Communications with the DIO port are very fast because there is no extra conversion circuitry needed for the output. The only ports that will be used in this project are the digital input/output ports A, B, and C. The encoders will be attached to and driven by these lines.

C. MICROSTRAIN SENSOR

1. Background Information

The MicroStrain sensor used in this project is the 3DM-GX1 Microminiature Sensor created by MicroStrain, Inc. in Williston, Vermont.[8] The sensor is a three degrees of freedom orientation sensor. It has a tri-axial angular rate sensor or rate gyro, three orthogonal magnetometers, three orthogonal accelerometers, and a temperature sensor. There is also a 16-bit analog to digital converter and an onboard microcontroller.[9] This sensor is capable of outputting orientation in both static and dynamic applications. The dynamic accuracy is considerably less than the static accuracy, as can be seen in the next section. A photo of the sensor can be found in Figure 6.

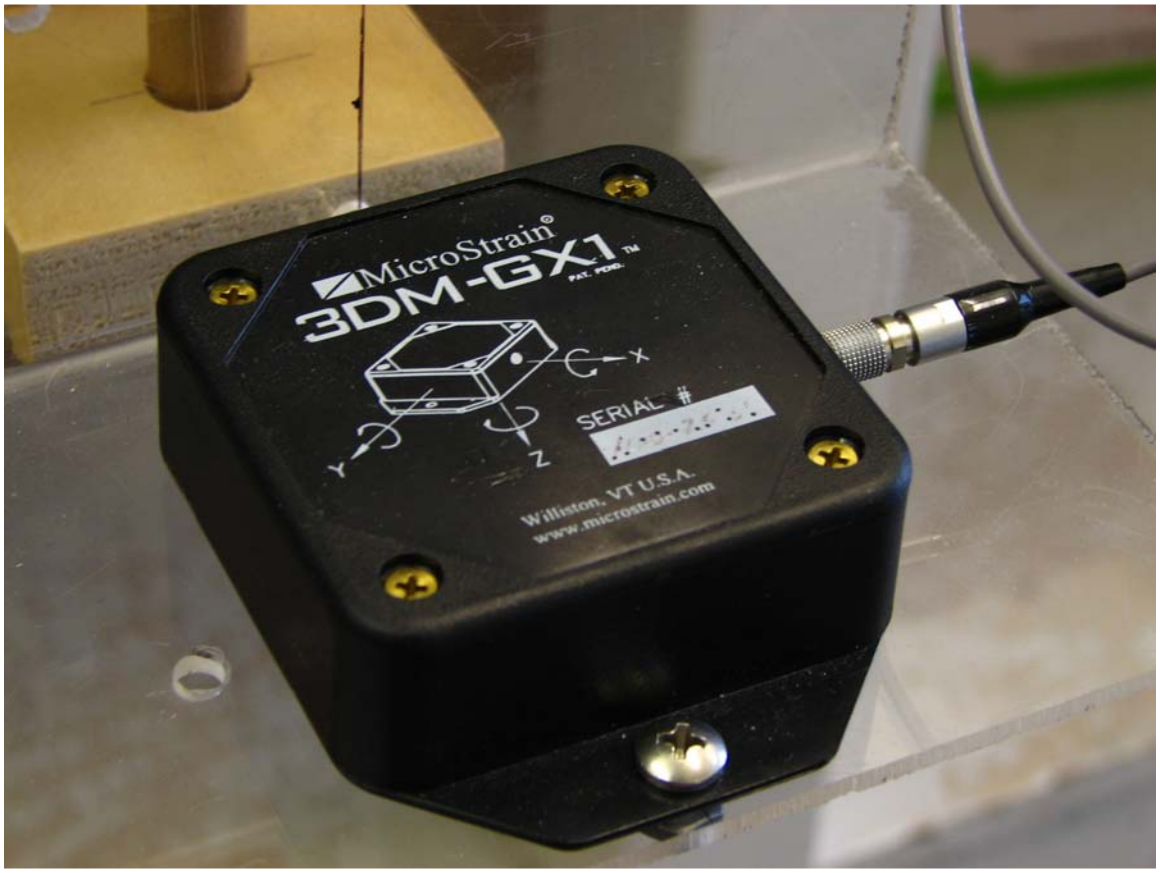
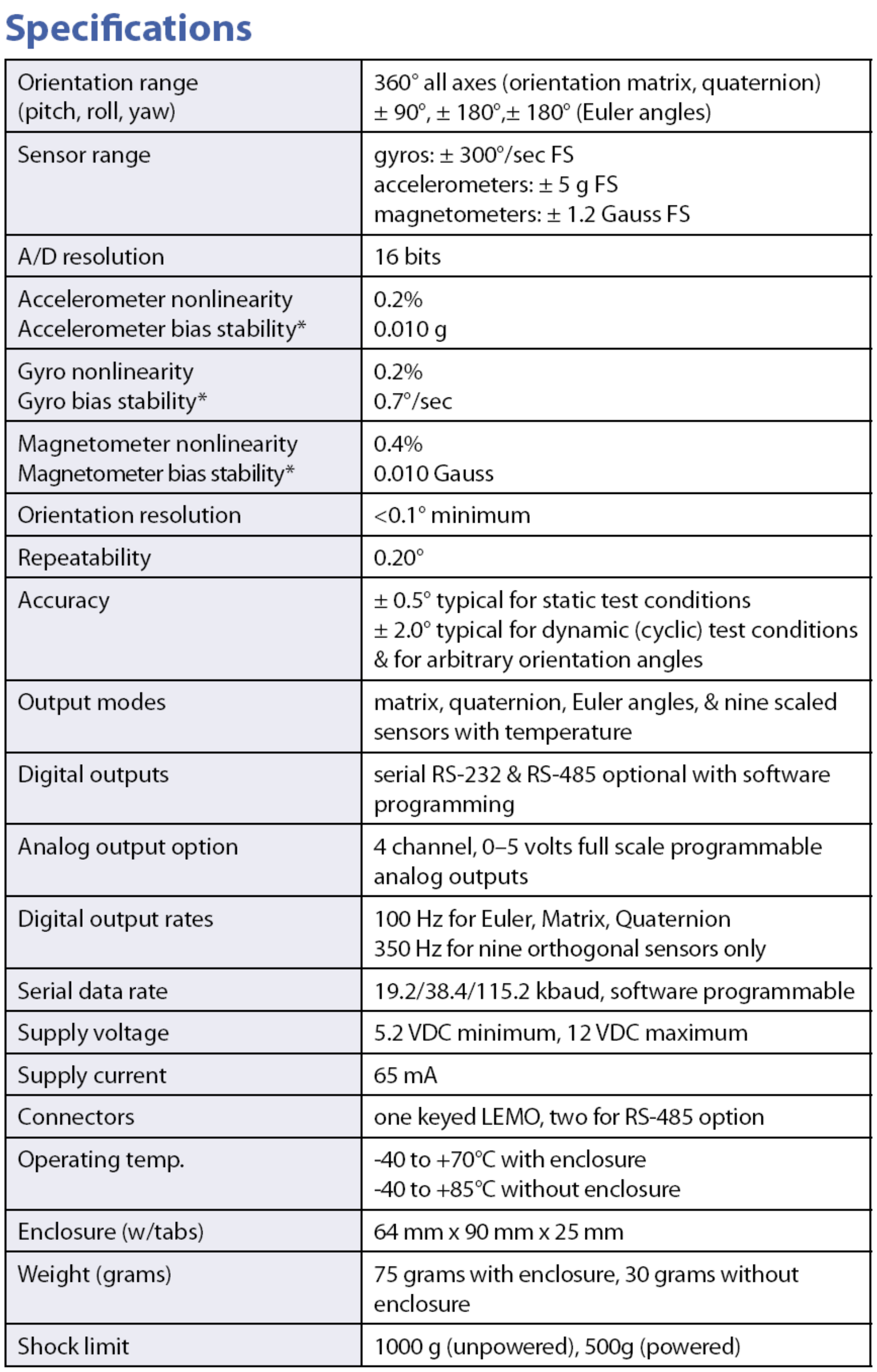


Figure 6. MicroStrain 3DM-GX1 Sensor.

The sensor is capable of RS-232 and RS-485 serial output. It is able to output raw data at a rate of 350Hz.[9] It also has the ability to first convert the raw data into more useful and recognizable forms such as Euler angles, matrix and quaternion before output. Drift in the sensor's data needs to be continually correct due to the error and drift of the gyro sensors. The error correction is calculated based on measurements from the accelerometers and the magnetometers. These two sets of sensors are more accurate under a static application and this characteristic is used to calculate the correction.[9]

2. Specifications

The following table is the specification table supplied by the MicroStrain company.



Specifications

Orientation range (pitch, roll, yaw)	360° all axes (orientation matrix, quaternion) ± 90°, ± 180°,± 180° (Euler angles)
Sensor range	gyros: ± 300°/sec FS accelerometers: ± 5 g FS magnetometers: ± 1.2 Gauss FS
A/D resolution	16 bits
Accelerometer nonlinearity Accelerometer bias stability*	0.2% 0.010 g
Gyro nonlinearity Gyro bias stability*	0.2% 0.7°/sec
Magnetometer nonlinearity Magnetometer bias stability*	0.4% 0.010 Gauss
Orientation resolution	<0.1° minimum
Repeatability	0.20°
Accuracy	± 0.5° typical for static test conditions ± 2.0° typical for dynamic (cyclic) test conditions & for arbitrary orientation angles
Output modes	matrix, quaternion, Euler angles, & nine scaled sensors with temperature
Digital outputs	serial RS-232 & RS-485 optional with software programming
Analog output option	4 channel, 0–5 volts full scale programmable analog outputs
Digital output rates	100 Hz for Euler, Matrix, Quaternion 350 Hz for nine orthogonal sensors only
Serial data rate	19.2/38.4/115.2 kbaud, software programmable
Supply voltage	5.2 VDC minimum, 12 VDC maximum
Supply current	65 mA
Connectors	one keyed LEMO, two for RS-485 option
Operating temp.	-40 to +70°C with enclosure -40 to +85°C without enclosure
Enclosure (w/tabs)	64 mm x 90 mm x 25 mm
Weight (grams)	75 grams with enclosure, 30 grams without enclosure
Shock limit	1000 g (unpowered), 500g (powered)

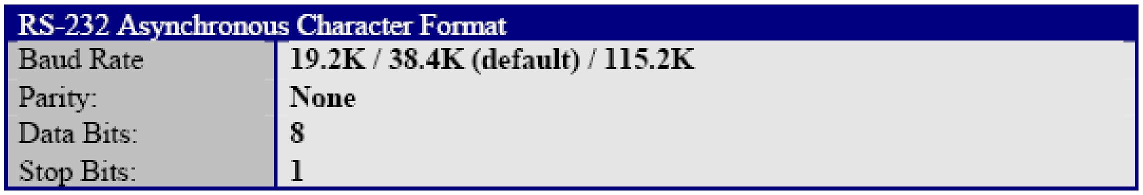
Table 1. MicroStrain 3DM-GX1 Specifications [From Ref. 9].

One of the main specifications that is very important to this project is the static and dynamic accuracy, which is seen in Table 1. The table indicates the static accuracy is ±0.5° and the dynamic accuracy is ±2.0°. This is a considerable difference. One of the main purposes of this project is to test this dynamic accuracy. If the sensor is moving quickly, such as in a human motion tracking application, the dynamic accuracy has a large impact. A larger uncertainty range will not allow accurate tracking and execution of a control routine.[9]

The sensor range in the table is an acceptable range. In a human limb motion tracking application 300° per second is more than enough range to capture the normal motion of a limb. In this project, the pendulum designed and created as part of the apparatus will not be traveling faster than this sensor range. It will also not be experiencing any motion that will create more than ±5G's of force.[9]

3. Communications

This MicroStrain sensor will be using the RS-232 serial communications standard for communications with the target PC104. The default communications protocol is RS-232 with 38.4kbps, no parity, one stop bit, and eight data bits, as seen in Table 2.



RS-232 Asynchronous Character Format	
Baud Rate	**19.2K / 38.4K (default) / 115.2K**
Parity:	**None**
Data Bits:	**8**
Stop Bits:	**1**

Table 2. 3dM-GX1 Default RS-232 Format [From Ref. 10].

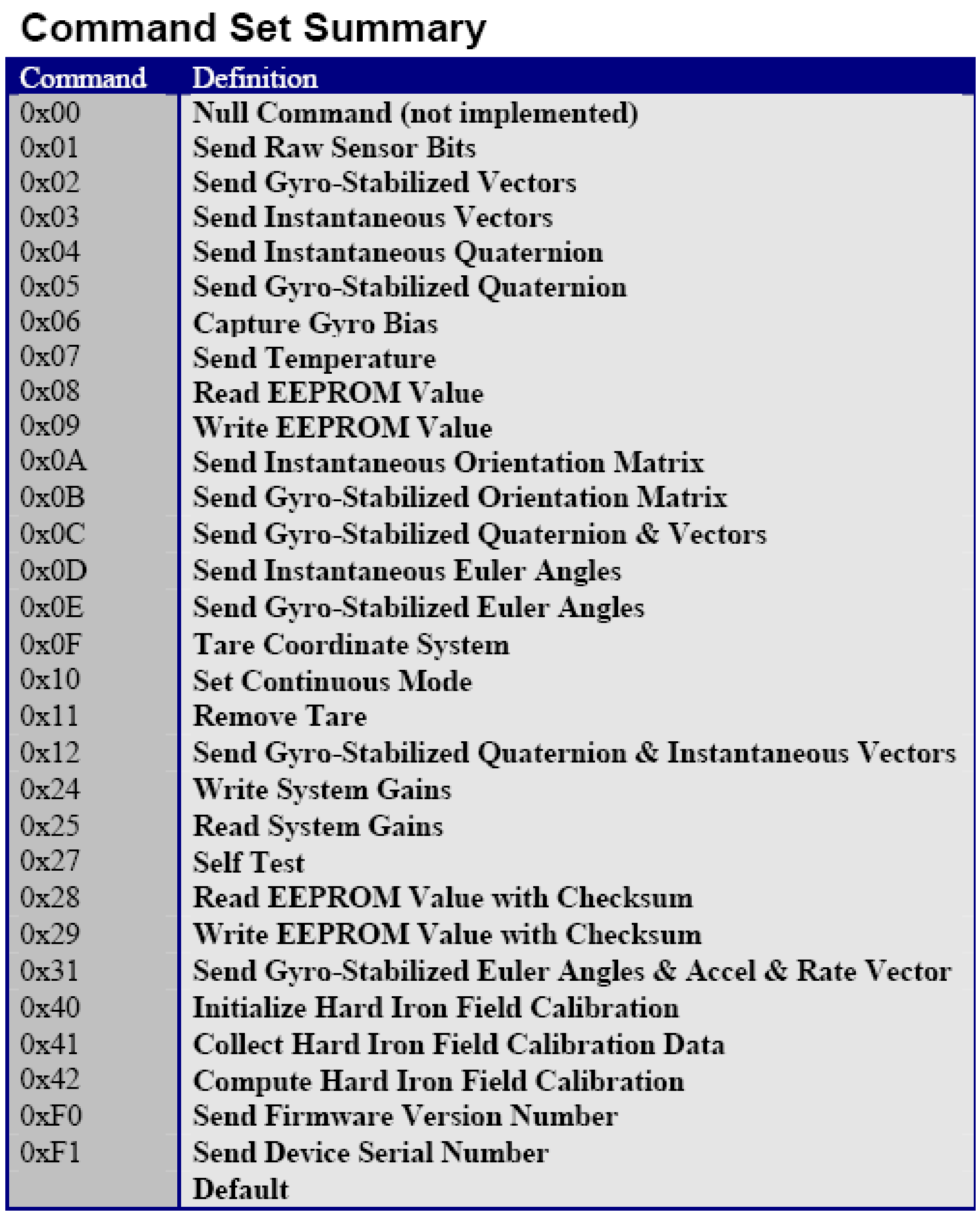
There are two basic modes of communications with the sensor, polled command mode and continuous mode. Continuous mode is where the sensor sends out the requested data continuously after the request by the host. A new data pack is transmitted by the sensor after every calculation. Calculations are continuously performed allowing the sensor to output the data at its maximum rate. The stream of data will have no gaps because the transmission and calculation is performed at a set rate by the microcontroller. The host computer must be able to receive and process data at this rate or faster because if the host computer is not able to keep up with the data rate then there will be data packets that are dropped. If data packets are dropped the process that is relying on the data from the sensor will not execute correctly. The user is able to exit this mode of operation by issuing a stop command.[10]

The second mode of communication between the host and the sensor is polled mode. This is the default mode the sensor is started in. It is possible to change this setting by writing to the EEPROM onboard the sensor. Polled mode is more interactive with the user. In this mode the host must send commands to the sensor to request data. The sensor will then send the data back to the host. This method has limited accuracy. When the host issues the request for data, the packet that is sent back is the packet that is being calculated when the request is received by the sensor. This means there is a window of uncertainty that spans the calculation cycle time, which is 13.107ms for the current configuration of the sensor.[11] This window of uncertainty is relatively large with respect to the acquisition time of the encoder. To get the most accurate time the exact calculation time must be known or the time between data calculation start and the encoder reading be minimized. While the polled method could allow encoder reading to be taken at the exact time the data calculation cycle starts there is no way to guarantee or control this. The continuous method does not make it possible to record encoder position at exactly the data calculation start time of the MicroStrain sensor but the time between the first data bit output of the sensor and the calculation start time is known and can be compensated for. This enables the possibility to match encoder position with MicroStrain position based on a time shift of the encoder data.[10]

A third mode of communication is also possible. It is polling while in continuous mode. While the MicroStrain sensor is in continuous mode it is possible to also poll the sensor for a different set of data. This may increase the calculation time required depending on the data that was requested. The first set of data that is sent back from the sensor to the host is the continuous mode data. The polling mode data requested then follows. The host must be able to distinguish between the two data packets.

4. Acquiring Data – Gyro Stabilized Euler Angles

Acquiring data is performed in two steps. The first step is the host must issue a command. The sensor will then return the corresponding data. The host may have to issue more than one command to achieve desired results. Once the sensor is powered up it will automatically start all raw data calculations and continue these calculations until power is lost. The default mode is polling mode. In this case only the command byte must be sent to the sensor. This tells the sensor which set of data is requested. The entire list of possible commands is seen in Table 3. When the sensor finishes the calculation it is performing, assuming it receives a command, it will return the data in a form specific to the requested data. This is explained later in this section. The MicroStrain sensor is capable of storing up to 15 commands in its buffer. They will be output in order.[10]



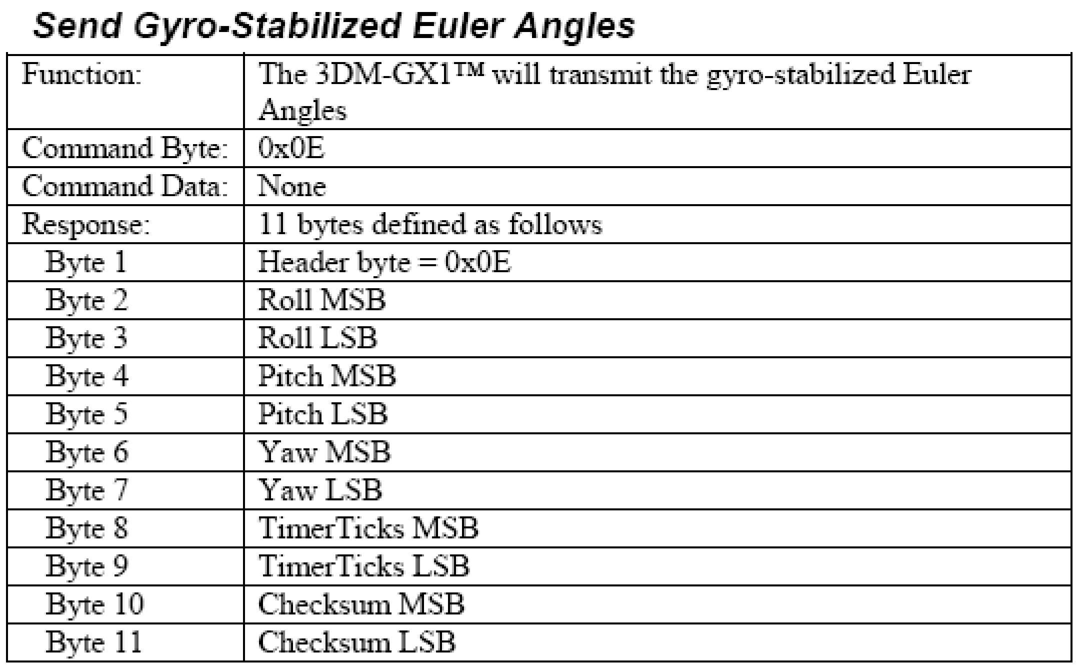
Command Set Summary

Command	Definition
0x00	Null Command (not implemented)
0x01	Send Raw Sensor Bits
0x02	Send Gyro-Stabilized Vectors
0x03	Send Instantaneous Vectors
0x04	Send Instantaneous Quaternion
0x05	Send Gyro-Stabilized Quaternion
0x06	Capture Gyro Bias
0x07	Send Temperature
0x08	Read EEPROM Value
0x09	Write EEPROM Value
0x0A	Send Instantaneous Orientation Matrix
0x0B	Send Gyro-Stabilized Orientation Matrix
0x0C	Send Gyro-Stabilized Quaternion & Vectors
0x0D	Send Instantaneous Euler Angles
0x0E	Send Gyro-Stabilized Euler Angles
0x0F	Tare Coordinate System
0x10	Set Continuous Mode
0x11	Remove Tare
0x12	Send Gyro-Stabilized Quaternion & Instantaneous Vectors
0x24	Write System Gains
0x25	Read System Gains
0x27	Self Test
0x28	Read EEPROM Value with Checksum
0x29	Write EEPROM Value with Checksum
0x31	Send Gyro-Stabilized Euler Angles & Accel & Rate Vector
0x40	Initialize Hard Iron Field Calibration
0x41	Collect Hard Iron Field Calibration Data
0x42	Compute Hard Iron Field Calibration
0xF0	Send Firmware Version Number
0xF1	Send Device Serial Number
	Default

Table 3. Command Set Summary for MicroStrain 3DM-GX1 [From Ref. 10].

To enter continuous mode the host must issue the "Set Continuous Mode" command, 0x10, followed by the desired data command, from Table 3. The sensor will respond by sending the desired data back to the user at a constant rate, depending on the setup of the sensor. This will continue until the sensor is reset or the stop command is issued. The stop command is similar to the command to enter continuous mode. The host must first issue the "Set Continuous Mode" command followed by the "Null Command", 0x00. This will cause the sensor to stop sending data out continuously. Continuous mode can be re-entered at any time if desired.

The command that is used in this project is 0x0E "Send Gyro-Stabilized Euler Angles." This command gives the roll, pitch, and yaw of the sensor with respect to the fixed earth orientation system. The angles are according to the ZYX or aircraft coordinate system[10]. The format of the returned data is seen in Table 4.



Send Gyro-Stabilized Euler Angles

Function:	The 3DM-GX1™ will transmit the gyro-stabilized Euler Angles
Command Byte:	0x0E
Command Data:	None
Response:	11 bytes defined as follows
Byte 1	Header byte = 0x0E
Byte 2	Roll MSB
Byte 3	Roll LSB
Byte 4	Pitch MSB
Byte 5	Pitch LSB
Byte 6	Yaw MSB
Byte 7	Yaw LSB
Byte 8	TimerTicks MSB
Byte 9	TimerTicks LSB
Byte 10	Checksum MSB
Byte 11	Checksum LSB

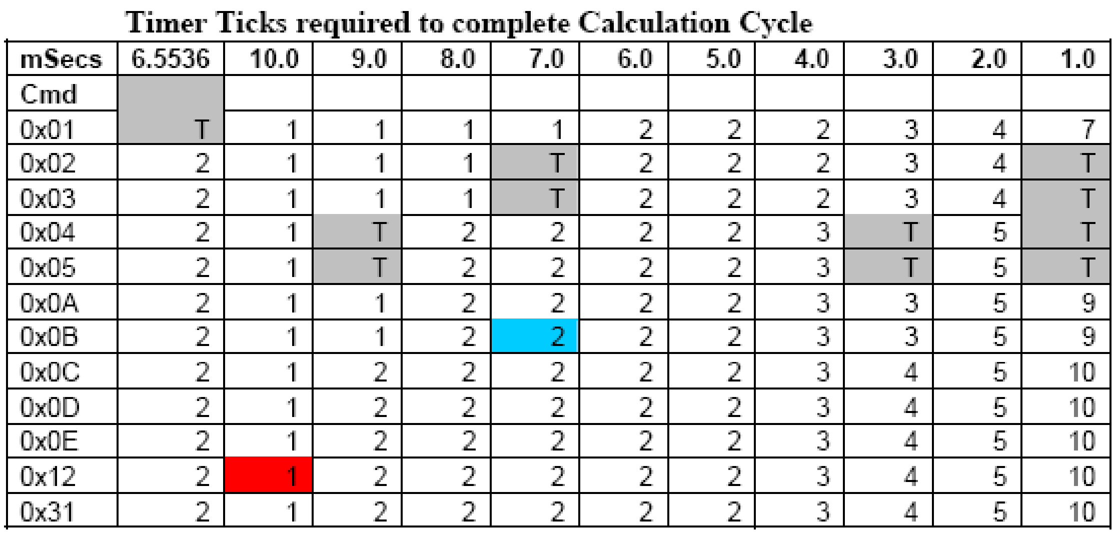
Table 4. Gyro-Stabilized Euler Angles Date Output Format [From Ref. 10]

The RS-232 receiver must be able to receive and process this data fast enough so that no data is lost. The sensor will continue to send data regardless of whether or not the receiver is receiving the data. The data in roll, pitch, and yaw is sent in a raw binary format. The roll and yaw have possible values ranging between -32768 to 32767, corresponding to -180° to 180°. The pitch has a range of -16384 to 16343, corresponding to -90° to 90°. These raw 16-bit numbers must be reformatted by a scaling them by 360/65536 to obtain the correct angle in degrees.[10]

The last four bytes of data sent is very important for error checking. This helps to ensure all data arrived and is valid. The "TimerTicks" is the word of data that holds the value of sensor clock ticks recorded at the beginning of every calculation cycle. "TimerTicks" is a 16-bit number so when it reaches 65535 it will rollover to 0 on the next clock tick. "Checksum" is the reference number for correct data. The host must add all preceding bytes of data received together. If this number is equal to the "Checksum" then the data received is correct. If the numbers are different, the data is invalid for a list of possible reasons: interference, incorrect data set, etc. In the case of command 0x0E, "Send Gyro-Stabilized Euler Angles", the "Checksum" would be compared to 0x0E+roll+pitch+yaw+TimerTicks.[10]

5. Calculation Cycle Information

The amount of time between clock ticks is programmable. The default value is 6.5536msec.[11] This the value used for this project. The clock will always cycle at this speed. Not all the calculations will be completed in one cycle of the clock. It may take as many as 10 clock cycles to compute the output depending on the desired output and the clock speed, as seen in Table 5. For this project, at 6.5536 msec per tick, it requires 2 complete timer ticks for the 0x0E command calculation to complete. As seen, it is standard for all the command calculation timer ticks at 6.5536 msec to be two ticks. The time could be changed to 10 msec, which would clearly speed up the sample rate, but if all the sensors are defaulted to 6.5536, it makes more sense to test at default rather than reset the internal circuitry of every sensor this apparatus will test.[11]



Timer Ticks required to complete Calculation Cycle

mSecs	6.5536	10.0	9.0	8.0	7.0	6.0	5.0	4.0	3.0	2.0	1.0
Cmd											
0x01	T	1	1	1	1	2	2	2	3	4	7
0x02	2	1	1	1	T	2	2	2	3	4	T
0x03	2	1	1	1	T	2	2	2	3	4	T
0x04	2	1	T	2	2	2	2	3	T	5	T
0x05	2	1	T	2	2	2	2	3	T	5	T
0x0A	2	1	1	2	2	2	2	3	3	5	9
0x0B	2	1	1	2	2	2	2	3	3	5	9
0x0C	2	1	2	2	2	2	2	3	4	5	10
0x0D	2	1	2	2	2	2	2	3	4	5	10
0x0E	2	1	2	2	2	2	2	3	4	5	10
0x12	2	1	2	2	2	2	2	3	4	5	10
0x31	2	1	2	2	2	2	2	3	4	5	10

Table 5. 3DM-GX1 Clock Cycles per Calculation Type [From Ref. 11]

In the above table, the "T" means that particular command under that associated time does not have a guaranteed number of ticks. This is because the timing is so close to the edge of a timer tick that it can be assumed it transitions for some time on either side of a timer tick. Some cycles may take only 1 tick while at other times it may take 2 ticks. The red and blue cells are highlighted for purposes of the documentation it was drawn from and they have no purpose here.

Timer Ticks x Timer Tick Interval = Total Time to Complete Calculation Cycle

mSecs	6.5536	10.0	9.0	8.0	7.0	6.0	5.0	4.0	3.0	2.0	1.0
Cmd											
0x01	T	10	9	8	7	12	10	8	9	8	7
0x02	13.107	10	9	8	T	12	10	8	9	8	T
0x03	13.107	10	9	8	T	12	10	8	9	8	T
0x04	13.107	10	T	16	14	12	10	12	T	10	T
0x05	13.107	10	T	16	14	12	10	12	T	10	T
0x0A	13.107	10	9	16	14	12	10	12	9	10	9
0x0B	13.107	10	9	16	14	12	10	12	9	10	9
0x0C	13.107	10	18	16	14	12	10	12	12	10	10
0x0D	13.107	10	18	16	14	12	10	12	12	10	10
0x0E	13.107	10	18	16	14	12	10	12	12	10	10
0x12	13.107	10	18	16	14	12	10	12	12	10	10
0x31	13.107	10	18	16	14	12	10	12	12	10	10

Table 6. 3DM-GX1 Total Calculation Time per Type [From Ref. 11]

The above table shows the total time for each calculation cycle in milliseconds. To arrive at these numbers the clock cycle time is multiplied by the timer ticks required from Table 5. The "T", red, and blue cells are the same as in Table 5. The green cells are the fastest calculation cycle time for each command. Again, this was not used because all the default values were used to standardize the calculations from command to command and sensor to sensor. The above table shows the calculation cycle time for the 0x0E command is 13.107 msec. This means every 13.107msec a new data point can be read. The sensor outputs data at 76.295 Hz.[11]

D. ENCODERS

There were two encoders used in this project. One was a 10-bit absolute encoder created by Computer Optical Products, Inc. The other encoder was a 16-bit absolute created by Gurley Precision Instruments.

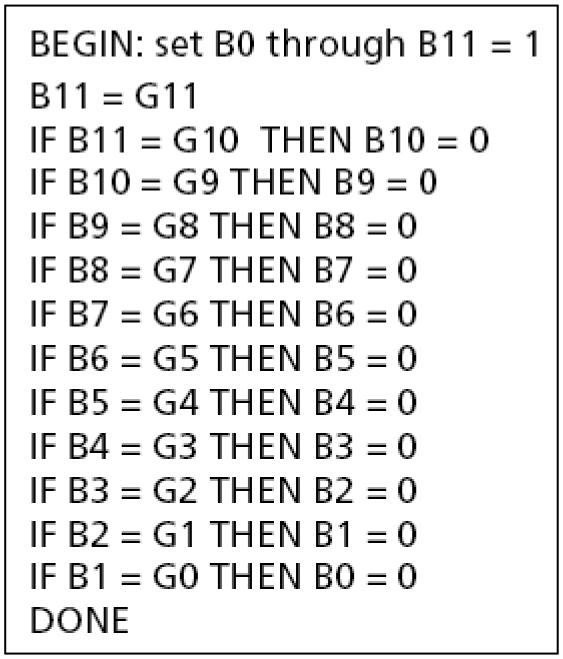
1. 10-Bit Encoder

The first encoder that was used in the project was the 10-bit absolute encoder, CP-350-10GC, manufactured by Computer Optical Products, Inc. The output is standard +5V TTL. 50kHz is the maximum data output rate. All data is output in a 10-bit parallel, grey code form. This requires a conversion from the grey code to binary code. Once the output is in binary code it can then be converted to an angular degree.[12]

There are a number of different methods that can be used to convert grey code to binary. The following general formula for the conversion is from Ref. 13:

$B_n = G_n \cdot B_n + 1 + G_n \cdot B_{n+1}$

In this project, a short code routine was used for the conversion. The pseudo code can be seen in Figure 7.



```
BEGIN: set B0 through B11 = 1
B11 = G11
IF B11 = G10  THEN B10 = 0
IF B10 = G9 THEN B9 = 0
IF B9 = G8 THEN B8 = 0
IF B8 = G7 THEN B7 = 0
IF B7 = G6 THEN B6 = 0
IF B6 = G5 THEN B5 = 0
IF B5 = G4 THEN B4 = 0
IF B4 = G3 THEN B3 = 0
IF B3 = G2 THEN B2 = 0
IF B2 = G1 THEN B1 = 0
IF B1 = G0 THEN B0 = 0
DONE
```

Figure 7. Grey Code to Binary Pseudo Code [From Ref. 13].

This code is clearly for a 12 bit application. For the purposes of this project the second line "B11=G11" would be changed to "B9=G9." All the lines after the second line would follow the pattern. The resulting binary number will be able to be converted to degrees by: DEGREES=BINARYNUMBER*360/1024.

2. 16-Bit Absolute Encoder

The 16-bit absolute encoder was created by Gurley Precision Instruments located in Troy, NY.[14] A photo can be found in Figure 8.

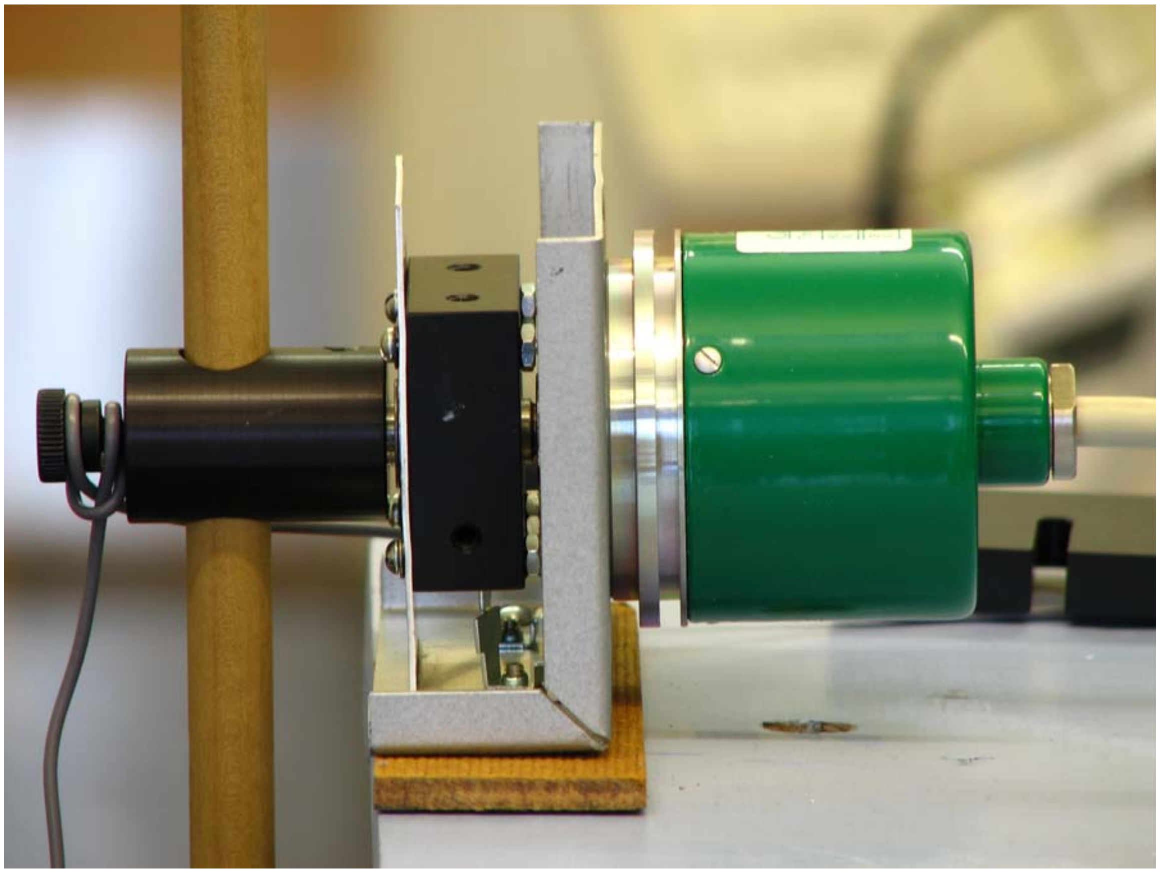


Figure 8. A58 16-Bit Encoder and Attachment.

It is a model A58 encoder. The part number is very important because it describes the encoder’s characteristics. It is A58S16MBTT05SAT39Q04EN. The following is the breakdown of the part number and characteristics[14]:

A58- Model number. It is specific to the Gurley Company.

S- Shaft type. S indicates a **s**olid shaft.

16- Resolution. It has **16**-bit resolution. The absolute resolution is 5.493×10^{-3} degrees per count.

M- Output Format. This encoder uses a **m**ultiplexed parallel output. 8-bits at a time are output, starting with the least significant 8-bits first. There are more specific output details later in this section. Serial output could have been selected but the DIO can read parallel much faster and more easily than serial. It only takes 2 read cycles to acquire all information with parallel communications while it would take 16 read cycles to get all the information with serial communication.

B- Output Code. The output code of this encoder is **b**inary. It can also be grey code like the 10-bit encoder. This is not practical for this project because the DIO ports are read and treated as a binary number. If grey code were used, a more time intensive program would have to be written to take the grey code and convert it into the corresponding binary number. This is an extra step that is eliminated if the encoder outputs the binary number to start.

TT- Output Device. Since this system is using digital input as the interface with the encoder it makes most sense to use a **TT**L signal coming from encoder. The other signal options would require extra circuitry to be compatible with the DIO input device.

05- Voltage. The digital to analog convert on the PC104 is able to output **05**.0 volts which can be used to drive the encoder.

S- Temperature Range. This system is not intended for implementation in hazardous situations so a **s**tandard temperature range of 0-70 C is sufficient.

A- Base type. **A** is the only option available for this model. It is a combination synchro flange/face mount base.

T- Cable Exit. The particular setup of this system would most easily interface with the encoder if the cable exited out the **t**op of the unit (back) of the encoder rather than the side.

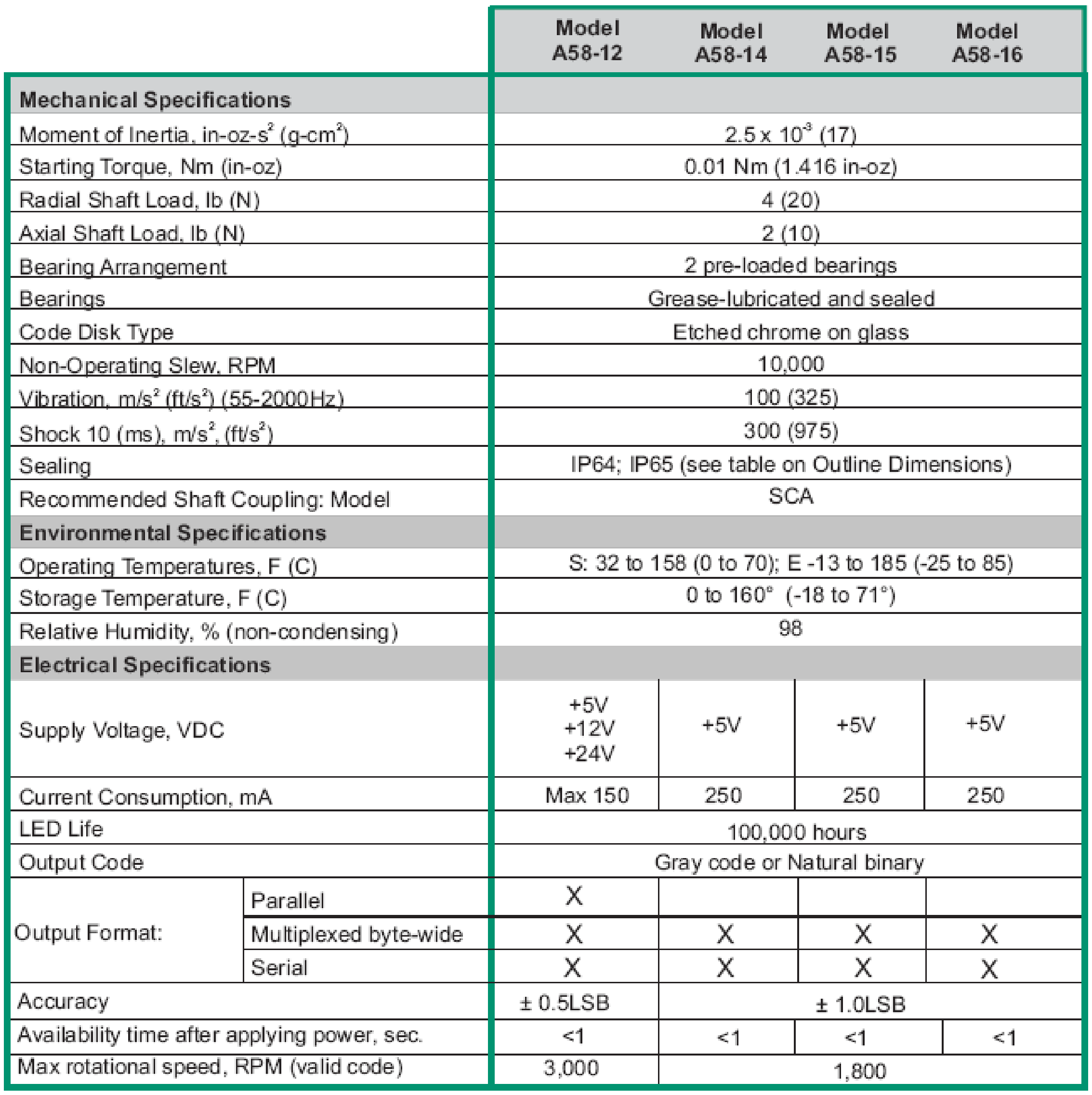
39- Cable Length in inches. A different size could have been selected, but **39**″ is the standard option so that is what was selected.

Q- Connector. **Q** corresponds to a DA-15P type plug. This is a 15-pin male connector. The pin assignment and plug diagram can be found later in this section.

04E- Shaft Diameter. **04E** corresponds to a shaft diameter of 1/4". This size was selected because the 10-bit encoder set up could already accept a 1/4" diameter shaft.

N- Special Features. There were **n**o special features ordered.

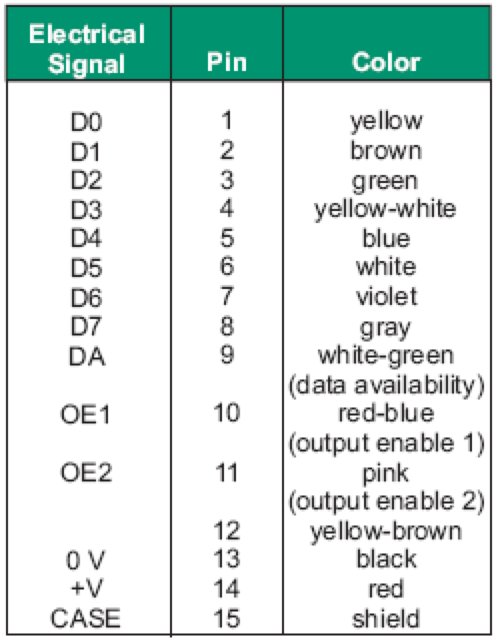
The overall specifications of the encoder can be found in Table 7. The far right column is the encoder used in this project.



		Model A58-12	Model A58-14	Model A58-15	Model A58-16
Mechanical Specifications					
Moment of Inertia, in-oz-s^2 (g-cm^2)		2.5×10^{-3} (17)			
Starting Torque, Nm (in-oz)		0.01 Nm (1.416 in-oz)			
Radial Shaft Load, lb (N)		4 (20)			
Axial Shaft Load, lb (N)		2 (10)			
Bearing Arrangement		2 pre-loaded bearings			
Bearings		Grease-lubricated and sealed			
Code Disk Type		Etched chrome on glass			
Non-Operating Slew, RPM		10,000			
Vibration, m/s^2 (ft/s^2) (55-2000Hz)		100 (325)			
Shock 10 (ms), m/s^2, (ft/s^2)		300 (975)			
Sealing		IP64; IP65 (see table on Outline Dimensions)			
Recommended Shaft Coupling: Model		SCA			
Environmental Specifications					
Operating Temperatures, F (C)		S: 32 to 158 (0 to 70); E -13 to 185 (-25 to 85)			
Storage Temperature, F (C)		0 to 160° (-18 to 71°)			
Relative Humidity, % (non-condensing)		98			
Electrical Specifications					
Supply Voltage, VDC		+5V +12V +24V	+5V	+5V	+5V
Current Consumption, mA		Max 150	250	250	250
LED Life		100,000 hours			
Output Code		Gray code or Natural binary			
Output Format:	Parallel	X			
	Multiplexed byte-wide	X	X	X	X
	Serial	X	X	X	X
Accuracy		± 0.5LSB	± 1.0LSB		
Availability time after applying power, sec.		<1	<1	<1	<1
Max rotational speed, RPM (valid code)		3,000	1,800		

Table 7. A58 16-bit Encoder Specifications [From Ref. 14].

The pin assignment of the encoder can be seen in Table 8.



Electrical Signal	Pin	Color
D0	1	yellow
D1	2	brown
D2	3	green
D3	4	yellow-white
D4	5	blue
D5	6	white
D6	7	violet
D7	8	gray
DA	9	white-green (data availability)
OE1	10	red-blue (output enable 1)
OE2	11	pink (output enable 2)
	12	yellow-brown
0 V	13	black
+V	14	red
CASE	15	shield

Table 8. Pin Assignment of Encoder Exit Cable [From Ref. 14].

Signals D0-D7 are data lines for the position data of the encoder. They carry the byte of data available based on the requested set. These lines are all output only. DA is the signal for Data Available and is active low. This signal is output only and it tells the user when the data on the data lines is valid. OE1 and OE2 are both input lines and are both active low. These lines indicate to the encoder when the user would like which set of data. Only one of these two lines should be low at a time. OE1 low indicates to the encoder the user would like the least significant byte of data. This is bits 0-7 of the encoder position. When OE2 is low the encoder outputs the most significant byte of data. This is bits 8-15. When both of these outputs are put together the 16 bit position of the encoder is realized. The 0V signal is the reference or ground signal for the encoder. This should be connected to the ground of the system. +V is the high voltage input. For the particular model used in this project the high voltage is +5V. CASE connects to the casement of the encoder and is not used in this project.[14]

The connector plug can be seen in Figure 9.

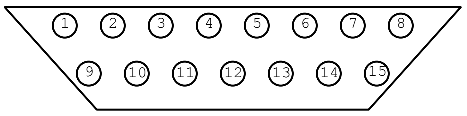


Figure 9. Male plug of the encoder.

The timing diagram of the encoder unit is very crucial in understanding how the encoder unit works as a whole. The timing diagram can be seen in Figure 10. The diagram shows lines OE1, OE2, DA, and D0-D7 with T1=80±10μs, T2≥300ns, T3≥300ns, and T4≤200ns.[14]

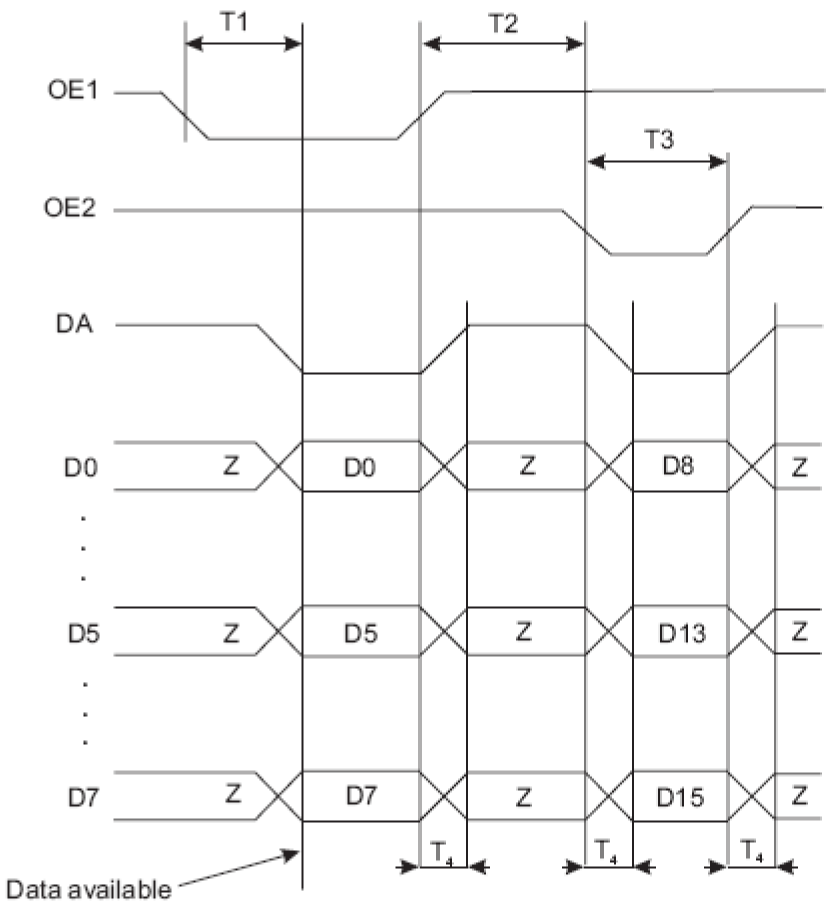


Figure 10. Timing Diagram of A58 Encoder [From Ref. 14].

These are the steps that must be followed to acquire data from the encoder:

1. OE1, OE2, DA are all high. D0-D7 are usually low, but this is not guaranteed. They are in an unknown state, Z in the timing diagram. When in this unknown state the data available on the lines is not valid.
2. User must take OE1 line low. The falling edge of OE1 latches the current position into the on board

memory. Every time this line is taken low, the encoder always clears the position memory and stores the current position.

3. After T1, DA appears low. The first instance that data is available is when DA is low. T1 is not too much of a concern in the project beyond being a reasonable time. The biggest concern is getting both sensors to latch at exactly the same time. The data read can occur any amount of time after the latch. A reasonable amount of time for T1 would be any time less than 100µs. This project does not require a degree of accuracy in dynamic motion faster than 10kHz because the MicroStrain sensor is not able to output faster than 350Hz.[11]
4. Once DA appears, the least significant byte may be read. D0 corresponds to bit 0 and D7 to bit 7 of the position.
5. After the read is complete the user must pull OE1 back to high to signal to the encoder the least significant byte read is complete. After T4 the data on the line will not be in any particular state. It is bad data and should not be read. T4 is not a crucial time but is included to indicate transition time for the data lines, even if the transition is to useless data.
6. After T2 the user must pull OE2 low. This indicates to the encoder to place the most significant byte of the already latched data on the data lines.

7. DA will then transition to low after T4. This indicates to the user that the upper byte of the position word is available.
8. The user is then able to repeat the process to latch and read another value. If OE2 is transitioned low before OE1 is transitioned low, the data available will still be the old data. OE2 may be pulled low and the upper byte data read any number of times without affecting the data. Although, as soon as OE1 is pulled low, the data is reset and a new set of data will be available.

If for any reason the V+ input does not have the correct voltage and current supplied, the output will not be correct. The encoder may appear to be functioning correctly at first, but upon further investigation, it will be found that the data output is faulty. The v+ of the encoder requires 5V at 250mA[14] and the data acquisition circuit is only able to supply 5V at 5mA of current from the DAC. This means the encoder must be driven by an outside source, or the constant +5v of the data acquisition circuit must be used. In this project the constant +5v from the data acquisition circuit was used.

E. THE PENDULUM

The pendulum is the main source of motion and connection between the sensors. The pendulum length is supplied by a half-inch wooden dowel rod which is 43 inches long. While the dowel rod does contribute more error due to torsion than metal would produce, the more important factor is the effect a metal rod would have on the internal magnetometers of the MicroStrain sensor. The MicroStrain sensor is connected to the wooden shaft by a plastic bracket. The combination of the mounting holes and the fact that the pendulum can rotate its orientation within the Z axis of the shaft allows the sensor to be placed in every vertical and upright horizontal orientation possible. The entire assembly can also be repositioned on its side to capture horizontal motion of the pendulum ensemble. The pendulum connects to the encoders by a clamp on the shaft. This ensures a strong secure connection but at the same time it allows the pendulum length to be readjusted. Readjusting the pendulum length will also allow the period to be altered. Since the MicroStrain sensor is intended to be used to capture human motion, having the adjustable period allows the pendulum to simulate different limb's sections performing different motions. Photos of the pendulum can be found in Figure 11.

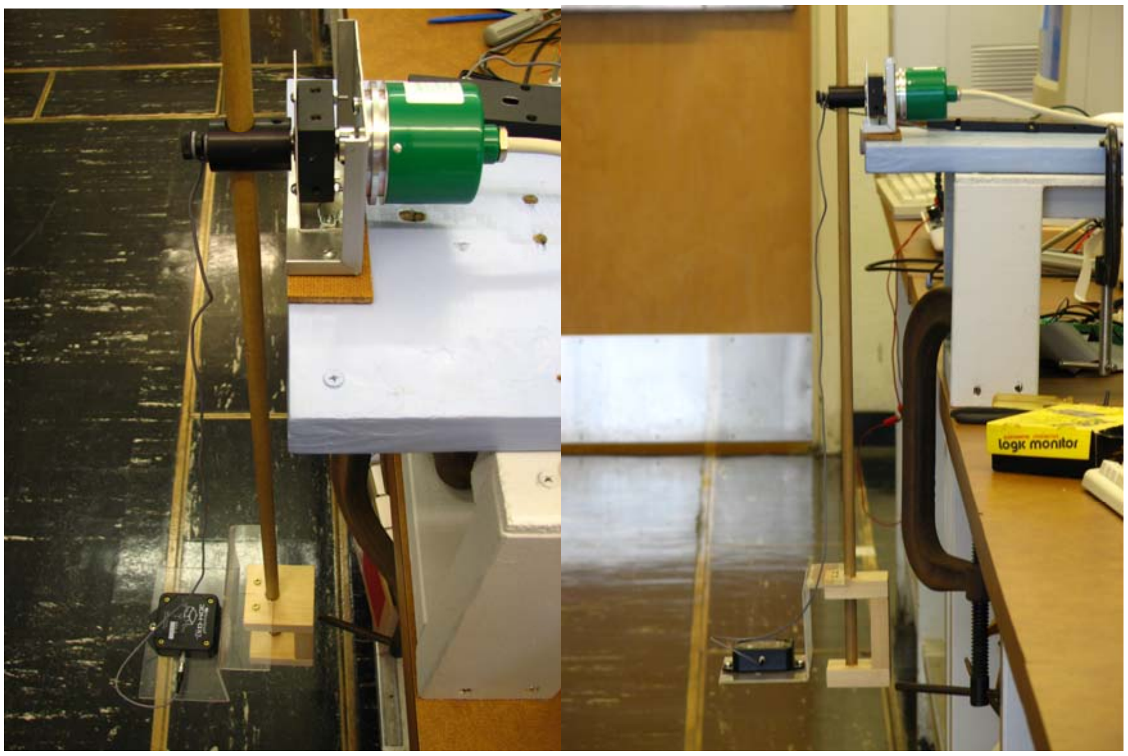


Figure 11. The Pendulum.

IV. TEST

A. SETUP

This section describes the setup of the system, concentrating on connections and inter-module communications. A block diagram of the test system can be found in Figure 12. The actual setup can be seen in Figure 13.

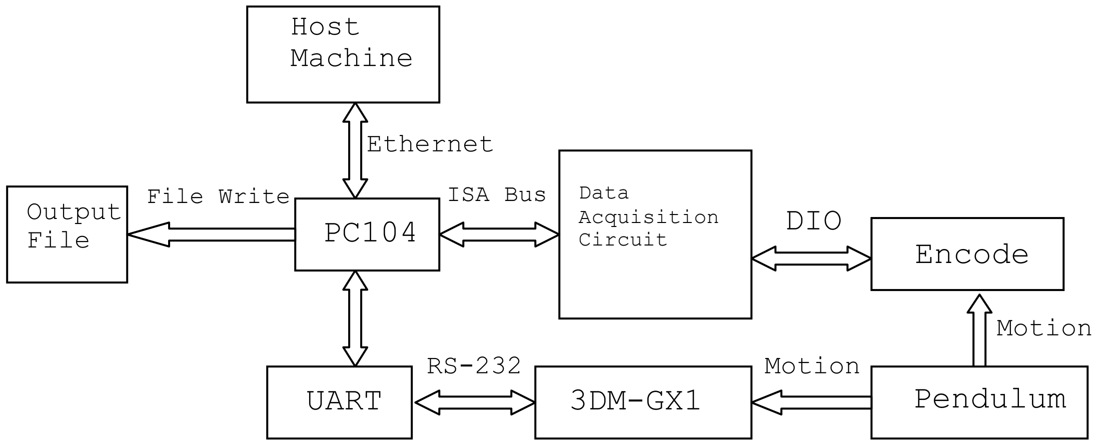


Figure 12. Block Diagram of Setup of test system.

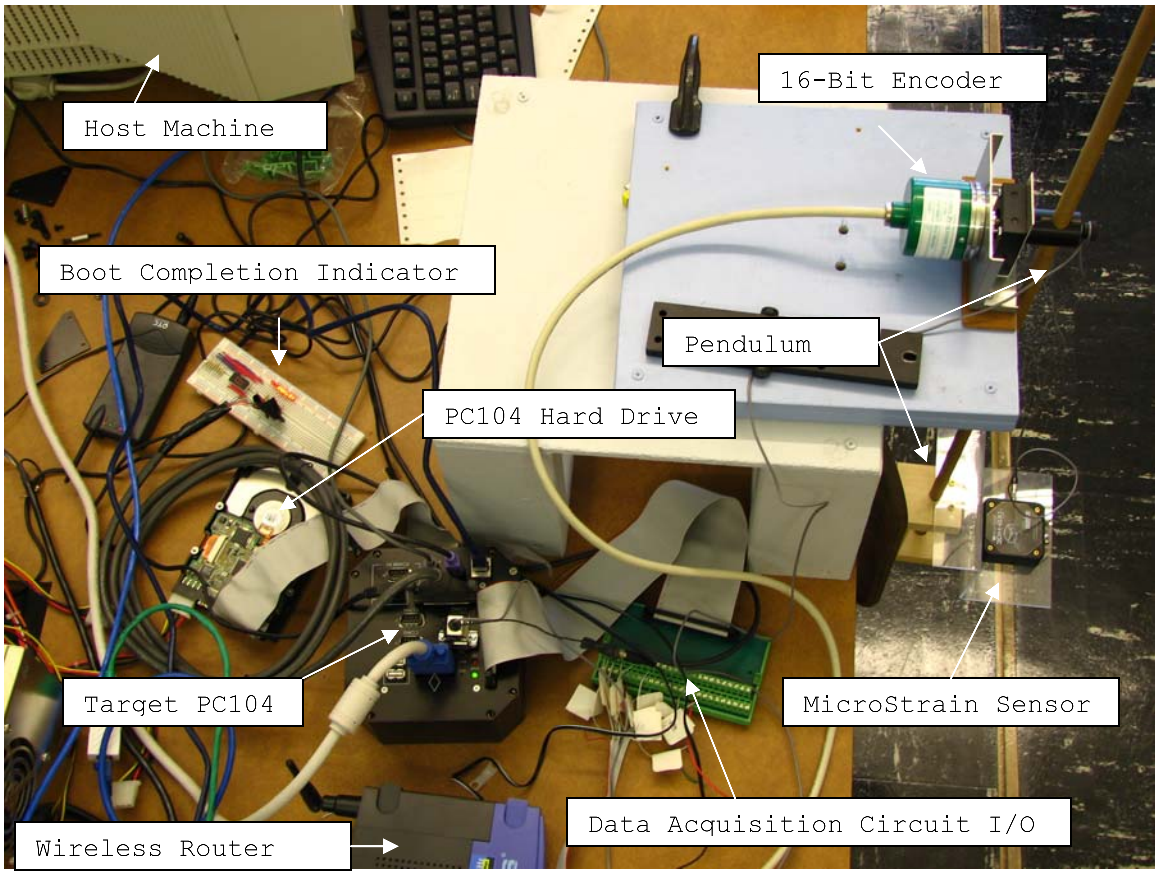


Figure 13. Actual Experiment Setup.

1. Ethernet and Associated Setup Procedures

The main form of communication between the host and the target of this system is an Ethernet connection. At different points throughout the project the two computers were either directly connected or connected through a wireless connection. The wireless connection used was the IEEE 802.11g. The host was connected directly into a wireless router and the target PC104 was connected to a wireless receiver. This connection was set up and used strictly for demonstration purposes. The connection was fast enough that program execution was not impeded. The only purpose of the connection between the host and target is for debugging, console out, and sending the compiled program to the target for execution. None of the programs that were tested used the console out in any important application. It was only used as a debug tool so the speed would have little impact on program execution.

For testing and data recording the hard connection between the host and the target was used. This consisted of a CAT-V crossover cable connection between the Ethernet ports of both systems. The connection speed used was 100Mbps. At this connection speed and under the demands of the program, there was little to no disadvantage of running under a host/target system rather than running strictly under a target only system.

For the Ethernet connection to be effective and for the communications to function properly, there were a few setup procedures that needed to be completed. The main communications between the two systems occurred because the QNX Momentics IDE needed to talk with the program executing on the target PC104. For this communication to be possible, the PC104 needs a small process to be continuously running that establishes this communication link. The program called *qconn,* developed by QNX, is the small process that allows this communication. *qconn* must be running on the target machine for Momentics to be able to send and execute compiled programs on the target machine. This process can be started at anytime after the OS is fully booted.

A second process must also be running in the background for proper communication between the host and the target. The process which allows communication between any two QNX based systems is called ***phrelay.*** This process allows remote login to occur from the QNX developed program Phindows. Phindows is a Windows based remote login program. It allows a user to start a totally new session of QNX Photon, the QNX OS graphical user interface. The session is hosted on the QNX system but is accessed and controlled by the remote Windows system. This program is an excellent tool that allows the PC104 to function as a very powerful remote embedded system. With this program the user is able to eliminate the need of a monitor or any direct physical contact with the PC104, yet at the same time have the capability to control every aspect of the machine. Remote file transfer protocol (FTP) communications are also necessary.

FTP allows the fast transfer of files between two computer systems. This is necessary for this project because the PC104 is not powerful enough to do much visual analysis of the recorded data, especially remotely. The MATLAB program is also not able to execute on the QNX OS. MATLAB is a very powerful data analysis tool that is used in this project. MATLAB will execute on a Windows system, thus creating the need for a file transfer. The output file that is stored on the PC104 that contains all the test data can be transferred to a Windows based system by an ftp file transfer.

The common UNIX program ***inetd*** gives the OS a way to manage most internet type services and processes. For the QNX OS to support FTP capabilities, ***inetd*** must be a currently running process. This allows a user to log into the FTP client of the QNX system from a Windows system. The command line FTP client on Windows systems is directly compatible with the QNX FTP client and is accessible when ***inetd*** is running. A second advantage of running this program is that ***phrelay*** is automatically started when the ***inetd*** process is started. When the ***inetd*** line is included in the /etc/rc.d/rc.sysinint system initialization file, the two processes will automatically start on operating system startup, allowing instant and automatic FTP and remote access client access from a remote system. The ***inetd*** line is included in this file directly following the system boot indicator program discussed previously.

2. RS-232

Serial Communication is the main form of communication between the MicroStrain sensor and the target PC104. Communication is performed following the RS-232 standard. The baud rate is 38.4Kbaud with 8 bits, 1 stop bit, and no parity. All communication with the sensor is directed through COM1 of the PC104, I/O address 0x3F8-0x3FF. The PC104 has a standard 16550 type UART on the micro CPU chip. The UART has a 16-byte first in first out buffer. The UART is set to IRQ 4. This means all interrupts associated with the UART will appear on interrupt request line 4. The specific setup of the COM1 port is discussed in the Acquisition Program section.

3. Digital Input/Output

Both encoders communicate with the PC104 through the digital input/output (DIO) ports of the data acquisition circuit. All the data is recorded as input to the DIO. The 16-bit encoder actually uses some digital outputs to drive the output enable lines. The signals are all TTL. 5V corresponds to logic 1 and 0V is logic 0. When the DIO ports are read, each port gives an 8-bit number representing the status of the port.

The 10-bit encoder has no control input so no digital output from the PC104 is necessary. The encoder is driven by the analog to digital conversion output "Vout0" on the data acquisition circuit, pin 29. The program outputs a constant 5V on this pin. This voltage drives the LED inside the encoder. The rest of the pin to pin connections can be seen in table 9.

Data Acquisition Circuit			Encoder	
	Pin		Pin	Grey Code Bit
DIO A0	1	←	8	G0
DIO A1	2	←	1	G1
DIO A2	3	←	12	G2
DIO A3	4	←	10	G3
DIO A4	5	←	11	G4
DIO A5	6	←	6	G5
DIO A6	7	←	3	G6
DIO A7	8	←	4	G7
DIO B0	9	←	2	G8
DIO B1	10	←	9	G9
Vout0	29	→	7	5V
Aground	33	↔	5	ground

Table 9. 10-Bit Encoder Pin-Pin Connection Table.

DIO ports A and B must be set as input. Port A when read will give the complete least significant grey code byte of data. Only the first two bits of port B must be read. If all the bits are read then only the first two must be used. These are the two most significant bits of the grey code position of the encoder. The two ground pins must be connected together to establish a common ground. If this is not connected then the DIO may not read the bits correctly and the LED in the encoder will not light.

The 16-bit encoder is connected in a slightly different configuration. The output enable control lines of this encoder require a 5V TTL signal. Originally, lines OE1 and OE2 were connected to an analog output of the data acquisition circuit. The effect of this is discussed in the Results section. The pin connection can be seen in table 10.

Data Acquisition Circuit			Encoder	
Description	Pin		Pin	Description
DIO A0	1	←	1	D0
DIO A1	2	←	2	D1
DIO A2	3	←	3	D2
DIO A3	4	←	4	D3
DIO A4	5	←	5	D4
DIO A5	6	←	6	D5
DIO A6	7	←	7	D6
DIO A7	8	←	8	D7
DIO B0	9	←	9	DA
DIO C0	17	→	10	OE1
DIO C1	18	→	11	OE2
+5V Out	29	→	14	5V
Aground	33	↔	13	ground

Table 10. 16-Bit Encoder Pin-Pin Connection Table.

Port A and B must be set for input. The entire byte of data is input through port A. Port B must also be set for input so “data acknowledge” can be read from the encoder. It is important for the computer to be able to read the DA signal. Port C must be set for output. Outputting OE1 and OE2 on a TTL line is crucial to get correct and accurate data. The encoder could be driven by a DAC of the data acquisition circuit at a quick glance, but in reality it is not possible. The DAC does not supply enough current to run the encoder without a buffer to supply the extra current. When the encoder power is connected to the +5V of the data acquisition circuit enough power is supplied to drive the encoder.

B. ACQUISITION PROGRAM FOR THE 16-BIT ENCODER

The acquisition program described here will be the program associated with the 16-bit encoder. The 10-bit encoder code is similar but the results and methods were not as accurate as with the following method used with the 16-bit encoder. The 10-bit encoder program will be described in a later section. It only describes the differences between the 10-bit encoder program and the following one.

The flow of the program is one of the most crucial aspects of the project. Incorrect programming could lead to inaccurate or incorrect data. The program is split into two main sections, the ***main*** routine and the interrupt service routine. The data acquisition board initialization, serial port setup and initialization, file output, and all other initializations will be described under the ***main*** function section while the DIO and serial port reading will be explained under the interrupt service routine section. Sections of code will be described in the following sections, although the entire program can be found in Appendix A.

The file *control.wse* is included in the beginning of the program. The file is a header file with function definitions that were used in the program. The file also contains all the header file inclusions that were used in the project. This slimmed down the program by writing all the include statements in a separate file. The *control.wse* file was added to for this project. The *base* variable is the base address of the data acquisition board and in this application it is 0x280. All registers of the data acquisition board are an offset of this address.

1. *Main* Function

The *main* function is where the program first starts execution. The first section of the *main* routine consists of mostly initialization routines. There are many initializations that must be completed for proper operation of the system. The first part of the code is seen here:

```
51     BoardInit();
```

All code here and in the following section, unless otherwise noted, is taken from Appendix A. This is the first few lines of the *main* function. *BoardInit()* is a function that is found in *control.wse*. The code for this function is seen here:

```
46 int BoardInit(void)
47 {
48     int privity_err;
49     privity_err=ThreadCtl( _NTO_TCTL_IO, NULL ); // thread gets
      root permission at access hardware
50     if(privity_err==-1)
51     {
52         cout<<"can't get root permission";
53         return -1;
54     }
55     out8(base + 0, 0x40); //reset the board, except the DAC output
56     return 1;
57 }
```

This code is taken from Appendix B. Line 49 is the most important line here. The *ThreadCtl* function is a QNX Neutrino specific command. It gives root permission for the thread to access all I/O functionality. With this line of code, the program will not be allowed to access any I/O functionality. Lines 50-54 simply return to the console an error message if root permission is not attainable. Line 55 is a data acquisition circuit specific register write. This line writes a 0x40 to the base address which causes the entire board to be reset. All DIO becomes input, all counters/timers are stopped, and all registers will be set to 0. The digital to analog circuitry is the only part of the board that is not affected. Once the board is reset the function will return a 1 if successful.

The next set of lines in the ***main*** function are seen here:

```
53      out8(base + 11, 0xFE); //set all DIOA DIOB to input DIOClow
        Output
54     SetOEBits(1,1);//OE1 OE2 high
```

Line 53 sets DIO port A and B to input, sets bits 1 and 4, and the lower nibble of port C to output, set bit 0. Bits 2,3,5,6,7 do not matter so they are set to 1, giving a value of 0xFE to be written to the register. When the DIO line control is changed from input to output for any of the ports, the respective ports are all pulled low. If the encoder is attached to the circuit and is currently on, the encoder will actually take a reading during this time. However, the flow of operations in this program do not allow this reading to affect any of the other readings. There must be careful attention given to this or the encoder may not produce the correct reading that is requested. The code for this function is seen here:

```
void SetOEBits(int OE1, int OE2)
{
    out8(base+10,(OE1 + 2*OE2)); //Pulls OE bits in either
  direction, 1-high, 2-low, OE1 must be on pin 17, OE2 pin 18
}
```

This function will receive two integers that must be either a 1 or a 0 for this function to work correctly. A 1 represents high and a 0 low. When the function is used, the values sent correspond to what the program requests the status of bits 0 and 1 of the DIO port C output be set to. Bit 0 is the first parameter and bit 1 is the second parameter. In the function, the only code executed is an output to register base + 10. This register is the DIO port C status register. Writing either a 0,1,2, or 3, as in this case, will change the bits 1 and 0 to represent the binary value of the number sent. This means that the other bits are all set to 0. In the case of this program there is no problem because none of the bits excluding 0 and 1 are used. If these bits are to be used in a different application, this code will have to be modified to first read the status of port C, change only bits 0 and 1, and then send the value back. In this program this code was not included because this function is executed in the interrupt service routine, ISR, as well as the ***main*** function. The idea is to minimize code execution in the ISR, therefore the extra code was decidedly excluded.

The next set of code is seen here:

```
56    SerialBit=0;
57    DataIndex=0;
58    DataSetSize=1000;
59
60    SerComInit();
```

Lines 56-58 initialize various variables for the program. This particular set of variables is actually global variables because they are used in both the ISR and the ***main*** function. They are initialized here to ensure proper operation. ***SerialBit*** is the variable that is used to indicate where in the receiving serial data packet the ISR is. This helps to ensure that only an 11 bit package is taken off the UART buffer at a time. More will be discussed on this in the ISR section. ***DataIndex*** is the current data set that is being recorded. This is used in the index that will indicate how many and where the data is currently being stored in the encoder data and MicroStrain sensor data arrays. ***DataSetSize*** is the desired number of data points to record. For this project, 1000 data points are enough data to allow a thorough analysis. Line 60 is the function that sets up and initializes the UART and RS-232 operation. Seen here is the actual function code:

```
void SerComInit(void)
{
    out8(sbase + 4, 0x09); //MCR data terminal ready
    out8(sbase + 3, 0x83); //enable latch for baud rate set
    out8(sbase + 0, 0x03); //lower baud divisor
    out8(sbase + 1, 0x00); //upper baud divisor
    out8(sbase + 3, 0x03); //disable latch, 1,N,8
    out8(sbase + 2, 0x07); //set FIOF to 1 character
    out8(sbase + 1, 0x15); //enable recv interrupt
}
```

A ***#define*** statement that establishes ***sbase*** is established at the beginning of the program code file. This is the base address of the UART. The address for the target PC104 is 0x3F8. All registers used for setup and initialization are offsets of this base address. There are many steps that are required to ensure proper operation of the UART and proper interface with the MicroStrain sensor. The modem control register is ***sbase*** + 4. This register control is used for handshake procedures. In the case of this program, there is no handshake needed because the sensor does not require any, so the port is set as follows:

Bit 0: 1 – Data terminal is ready for data

Bit 1: 0 – No request to send data

Bit 2: 0 – Option 1, not needed

Bit 3: 1 – Option 2, needed to enable proper interrupt operation on some PC based systems.

Bit 4: 0 – No loopback, normal operation

Bit 5: 0 – Not used

Bit 6: 0 – Not used

Bit 7: 0 – Not used[15]

The next register that requires setup is ***sbase*** + 3. This register is the line control register and is set up as follows:

Bit 0: 1 – Word length for traffic is set to either 6 or 8. Bit 1 specifies which.

Bit 1: 1 – Word length of 7 or 8. In conjunction with bit 0 the word length will be 8. This is needed because the MicroStrain sensor outputs 8 bit words.

Bit 2: 0 – Number of stop bits. The MicroStrain sensor outputs 1 stop bit. This bit must be set to receive and send 1 stop bit. Since the word length is set to 8 bits, a 0 in this bit will indicate 1 stop bit.

Bit 3: 0 – The MicroStrain sensor does not include any parity in its formatting. Setting this bit to 0 indicates no parity.

Bit 4: 0 – Not needed because parity is set to 0. This bit would otherwise indicate even or odd parity.

Bit 5: 0 – Not needed because parity is 0.

Bit 6: 0 – A 0 does not require a break condition. A break is not need for this application.

Bit 7: 1 – This gives access to the baud rate counter latch. If the baud rate is to be set, this bit must be set to 1. The baud rate is actually stored in a register that is mapped to two different locations. The transmit/receive buffer and the baud rate divisor least significant byte are both mapped to register offset 0 while interrupt enable and the divisor most significant byte are both mapped to register offset 1. When this bit is set, the divisor registers are the registers that are accessible while the others are not, until this bit is set back to 0. It is crucial for proper operation that this bit is set back to 0 before the UART is enabled for operation.[15]

Since the UART is now mapped to set the baud rate it would make most sense to set the baud rate. Lines 96 and 97 set the baud rate. The UART has a clock that is set to run at 1.8432MHz. To get the desired clock rate a divisor must be used to divide the clock rate to the desired baud rate.

In this case a baud rate of 38.4kbaud is needed. Under the 16550 standard of UART the 1.8432MHz clock is naturally divided by 16 before the divisor is applied. This means the clock is 115.2Kbaud if the divisor is 1. A divisor of 3 will result in a baud rate of 38.4Kbaud. This value is entered into the least significant byte of the baud rate divisor, ***sbase*** + 0, line 96 of the code. The most significant byte of divisor is 0, which is set in line 97. Immediately following this instruction, the line control register, ***sbase*** + 3, is again set in line 98. This resets only bit 7 of the register. It is very important to ensure all other bits are unaltered as this would change the parity and the word format. Line 98 writes 0x03 to the register. The only difference between this instruction and the instruction previously written in line 95 is bit 7 is changed to a zero. This will allow the UART to remap the 0 and 1 register offsets back to the transmit/receive buffer and the interrupt enable register.

Lines 99 and 100 set up the first in first out buffer control register and the interrupt enable register. Since the MicroStrain sensor sends data back at the end of its calculation cycle in a packet of data that has a size that varies with the type of data, the interrupt will need to be set when it receives as little as 1 character. If the interrupt is called for every character, it does not matter the size of the data packet being sent by the sensor because the ISR will be able to take all the characters sent one at a time. The ISR will have to be told how large of a data packet to expect to ensure each character is placed within the correct data packet.

It is very important to synchronize the encoder measurement and the MicroStrain sensor measurement as accurately as possible. One way for the sensor to communicate with the encoder is through the interrupt service routine. If the encoder is read on the last data character sent by the sensor, it will be possible to synchronize the two measurements as accurately as possible. The data packet sent by the sensor in this project is 11 characters long. The first in first out character buffer of the UART can cause an interrupt on 1,4,8, or 14. The only value that works for the application in this project is the 1 character setting. This setting, in conjunction with the ISR knowing which character of the data packet it is currently receiving, allows the ISR to know exactly when to take an encoder reading, thus minimizing the time difference between the encoder reading and the MicroStrain sensor reading.

The first in first out (FIFO) buffer control register is initialized in line 99 of the code. The register offset is 1. The bits are set as follows:

Bit 0: 1 – Enable transmit/receive FIFO.

Bit 1: 1 – Clear the contents of the receive FIFO. After this is called the value of this bit will be automatically reset to 0.

Bit 2: 1 – Clear the contents of the receive FIFO. After this is called the value of this bit will be automatically reset to 0.

Bit 3: 0 – No change in the transmit/receive mode. The difference in mode only deals with how the

RXRDY and the TXRDY pins function. These pins are not used in this project so this bit will remain unchanged.

Bit 4: 0 – Not used.

Bit 5: 0 – Not used.

Bit 6: 0 – Trigger level for the receive FIFO interrupt. A 0 here and in bit 7 will indicate the level 1.

Bit 7: 0 – In conjunction with bit 6 being a 0, the FIFO will cause an interrupt every time there is 1 character in the buffer.[15]

In order for the interrupts to actually occur the interrupt enable register must be set. Line 100 contains the instruction that will perform this task. The register offset of 1 is set with the following byte:

Bit 0: 1 – Enable the receiver ready interrupt. This interrupt will occur if the FIFO has a character that is ready to be received.

Bit 1: 0 – Disable the transmitter empty interrupt. Transmit interrupts are not useful in this application

Bit 2: 1 – Receiver line status register change interrupt. This interrupt will occur for various reasons. Data available, parity error, framing error, and various other errors will cause this interrupt. If this interrupt occurs the line status register, offset 5, will have to be polled to determine the cause of the interrupt. This interrupt is never actually used in the code for the project, but this interrupt could be used. The ISR will discard any information that is not correct based on what data is expected.

Bit 3: 0 – Modem status register interrupt. This interrupt is not needed since the only program that will be altering the UART is this program.

Bit 4: 1 – Sleep mode for a 16750 UART only. This does nothing in this project.

Bit 5: 0 – Not used.

Bit 6: 0 – Not used.

Bit 7: 0 – Not used.[15]

Writing the previous byte to ***sbase*** + 1 initializes and starts the indicated interrupts. Any information received by the UART will now cause an interrupt. The next step of the program is to ensure the MicroStrain sensor is not outputting any data and it is not in continuous mode before the interrupt service routine is enabled. If it were in continuous mode, the interrupts would start as soon as the interrupt service routine is attached to the interrupt of the UART. To maintain full control over the interrupts the sensor must not be sending data back until the program is ready to receive the data. Line 61 of the code, under the ***main*** function, is the function that will do this:

```
    StopCONTmode();
```

The definition of this function is after the *main* function and seen here:

```
123 void StopCONTmode(void)
124 {
125     out8(sbase + 0, 0x10); //Command Command
126     out8(sbase + 0, 0x00); //Null Command
127     out8(sbase + 0, 0x00); //Null Command
128 }
```

To output characters on the serial line, COM1 serial port, the characters must be written to the transmit FIFO buffer. This is accomplished by writing to the base register of the UART, address *sbase*. When the character is read it is then sent out by the UART. Line 125 sends the “Set Continuous Mode” command over the RS-232 line. This is then followed by two null commands in line 126 and 127. This sequence of characters, when received by the sensor, indicates that the sensor is to stop continuous mode and listen for a further command, either a polling command or another command to reenter continuous mode. Polling and continuous modes are explained in more detail in the MicroStrain sensor section of this paper.

Continuing in the *main* function, the next command is the interrupt initialization function:

```
    ISR4Init();
```

This function initializes and attaches the interrupt service routine to the interrupt of the UART. This interrupt is set to occur on IRQ line 4. This is set in the BIOS when the target PC104 first starts to boot. The following code is from the *control.wse* file:

```
/* Initialize IRQ 4 handler*/
int ISR4Init(void)
{
        isr4handid=InterruptAttach(4,isr4_handler,    NULL,    0,
  _NTO_INTR_FLAGS_TRK_MSK);
}
```

ISR4Init() initializes the interrupt service routine for IRQ 4. Line 364 attaches the interrupt handler ***isr4_handler*** to IRQ 4. Once the handler is attached the returned value is the ID of the ISR handler. This value is only used to mask and unmask the interrupt. ***isr4handid*** is a global variable that is declared at the beginning of ***control.wse***. The ISR handler code follows; it also is contained in ***control.wse***:

```
354 /*IRQ 4 Handler*/
355 const struct sigevent* isr4_handler(void *arg, int intr)
356 {
357     isr4_routine(); //ISR routine
358     InterruptUnmask(4,isr4handid); //Enable IRQ 4
359 }
```

The QNX operating system uses signals to communicate interrupts. This is why the type definition of the interrupt handler is ***sigevent***. The first line of the handler is the interrupt service routine. The code for the ISR is found in the main code, just above the ***main*** function. Every time an interrupt occurs on IRQ 4, this ISR handler is executed, which in turn executes the ISR. The last line of the interrupt handler, line 358, is the unmask command. This command tells the operating system to unmask all interrupts on IRQ 4 that are attached to the interrupt handler with the id in ***isr4handid***. This is the final instruction that actually stats the action of the interrupts to be handled by the defined ISR. After this command all interrupts on IRQ 4 will cause the ISR to execute.

After all the initializations are complete, the program is ready to start collecting data. This is started in line 64 of the code:

```
64     SendInstantEulerAngleCONT();
```

The definition of this function can be found later in the code file, and seen here:

```
116 void SendInstantEulerAngleCONT(void)
117 {
118     out8(sbase + 0, 0x10); //Command command
119     out8(sbase + 0, 0x00); //Null command
120     out8(sbase + 0, 0x0E); //Send GYRO Euler Angles command
121 }
```

This is the command that actually starts the data collection. The interrupt service routine is ready to execute, the serial communication is set up, and the DIO is ready to record. Line 118 is the "Set Continuous Mode" command. As indicated previously, this alerts the MicroStrain sensor to change the output mode based on the next two characters received. If the mode needs to be changed, a null command must follow the command. The sensor now knows that it must continuously output the data the next command received indicates. Line 120 is the command for the gyro-stabilized Euler angles. This command indicates to the sensor to continuously compute and output the gyro-stabilized Euler angles of the sensor. As soon as the sensor takes the reading and sends the first character of data back over the RS-232 line, the UART receives the character and flags an interrupt, indicating the buffer has a character to be taken.

Once the sensor is outputting data and the interrupts are occurring, the only task the `main` routine is responsible for is letting the user know the program is still executing. This is performed by the following code:

```
    while(DataIndex<DataSetSize)
    {
        if(checksum==dataid+roll+pitch+yaw+timerticks)
        {
            cout<<"Its good data"<<endl;
        }
```

```
72        else cout<<"BAD DATA!!!!"<<checksum<<"
      "<<dataid+roll+pitch+yaw+timerticks<<endl;
73    }
```

This is simply a ***while*** loop that will loop until enough data has been collected by the interrupt service routine. The *DataIndex* variable is incremented every time the interrupt service routine has received a full set of valid data. For this application, a valid set of data is a set of 11 characters from the MicroStrain sensor that represent one set of Euler angles. The ***checksum*** is a global variable that is used to verify the correctness of the data set. This procedure is explained in the MicroStrain section of this paper. The ***if*** statement verifies if the most recent data collected is valid. If the data is valid the program will output to the console "Its good data." This indicates to the user that the program is still operating and the data being collected is valid. If the data is not valid, meaning all the characters collected do not equal the checksum when added together, then the console will output "BAD DATA!!!!" A bad data output to the console could mean the data is bad for various reasons. The checksum and the character addition are also output to the console if the data is invalid. The user is then able to verify why the data is invalid. When the ***while*** loop is exited, the ***main*** function is responsible for stopping all operation and detaching the interrupt.

```
76    StopCONTmode();
77    InterruptDetach(isr4handid);
78    FileOutput();
```

As explained previously ***StopCONTmode()*** sends commands to the MicroStrain sensor that halts all continuous output. This will also cease the interrupts from occurring on IRQ 4. Line 77 detaches whichever interrupt handler id is sent to it. In this code it is the handler id assigned to ***isr4handid*** when the interrupt was initialized. The final major task performed in the main function is ***FileOutput().*** This function outputs all the data for each sensor in a file that is capable of being read by MATLAB. The function definition is seen here:

```
130 void FileOutput(void)
131 {
132     float myoutf;
133     FILE* myfile;
134 
135     myfile=fopen("NewEncoder23.m","w+");
136     fprintf(myfile,"Encoder=[");
137     for(int i=0;i<DataSetSize;i++)
138     {
139         fprintf(myfile," %d\n",EncoderValue[i]);
140     }
141     fprintf(myfile,"]';");
142 
143     fprintf(myfile,"\n\nMicroStrain=[");
144     for(int i=0;i<DataSetSize;i++)
145     {
146         myoutf=((float)rollnums[i])*360.0/65536.0;
147         fprintf(myfile," %f\n",myoutf);
148     }
149     fprintf(myfile,"]';");
150 }
```

This starts by creating a new file. Line 135 opens a new file with the name in the string parameter. The file must end with an ".m" to be directly compatible with MATLAB. The "w+" parameter creates a new file if a file with the given name does not exist. If the file exists, the file is opened and its contents are erased. In either case the file is opened for reading and writing. In this code only a write operation is performed. ***myfile*** is an output stream leading to the file indicated. The information from the sensors must be output as an array for easy import into MATLAB. This means the array must be declared in the file in such a way that MATLAB will understand. Line 136 is the start of the array. The string is sent to the ***myfile*** stream which will write it to the file. The ***for*** loop continues to output the encoder data from the encoder data array on the computer. When the data is completely transferred to the file, a "]';" must be output to indicate the end of the array for MATLAB. The function continues by outputting the MicroStrain sensor data in the same fashion. The only major difference is the inclusion of line 146. This instruction converts the raw data of the sensor into a usable angle. Since there are 360° in a circle and the sensor has 16-bits of accuracy, the raw data must first be multiplied by 360 and then divided by 2^{16} or 65536. The Euler angle is then output to the file. When the encoder data was output in the same fashion the data was truncated to the 10th's place for an unknown reason. This is the reason why only the MicroStrain sensor data was output. When the data is imported into MATLAB, the encoder data must be converted to angles in the same fashion for an accurate analysis between the two sensors.

The final lines of the ***main*** function output "Done" to the console to indicate to the user the data has been written to the file, the program is finished executing, and the data may be analyzed by MATLAB. The last line returns a 1 to indicate successful operation of the function. This is only included because ***main*** is of type ***int.*** The code is seen here:

```

    cout<<"Done";
    return 1;
}
```

2. The Interrupt Service Routine

The interrupt service routine is the function that is executed every time the associated interrupt occurs. It is executed by the interrupt handler. This sequence of events is discussed in the previous section. It is very important that the interrupt service routine, ISR, execute as quickly as possible. No extraneous code or unneeded processing must be performed in the ISR. It has been discovered that the interrupt service routine, under the specific conditions, operating system, and PC104 used in this project, must not contain any floating point operations. Floating point operations cause the PC104 and the QNX OS to lock up and a hard reset is needed to continue any further use of the computer. The reason may be that floating point operations are very time intensive causing the interrupt service routine execution time to be large enough that a new interrupt interrupts its own interrupt service routine.

The basic function of the ISR is to read characters off the UART receiver buffer, read the encoder after the last character is received, piece together the corresponding data from the UART and finally save all data in arrays for each sensor.

The first lines of the code are here:

```
int isr4_routine()
{
    if(SerialBit<11) SerialBuffer[SerialBit]=in8(sbase + 0);
  //read a character off buffer
```

The basic idea here is that if the index of the character being received from the UART buffer is less than 11, meaning the full data packet has not yet been received from the MicroStrain sensor, the character recorded in the

SeriaBuffer array, at the appropriate index. When a character is read off the buffer the UART interrupt is reset and waits for another character. Continuing the routine:

```
  if(SerialBit==10) //Get all Encoder Data and calc roll pitch and yaw
  {
      SetOEBits(0,1);//OE1 Low
      while(in8(base + 9) & 1){}
      SetOEBits(1,1);//OE1 High
      //1.8us
      SetOEBits(0,1);//OE1 Low
      while(in8(base + 9) & 1){}
      EncoderValue[DataIndex]=in8(base + 8); //read LSB
      SetOEBits(1,1);//OE1 High
      //1.6us
      SetOEBits(1,0);//OE2 Low
      while(in8(base + 9) & 1){}
      EncoderValue[DataIndex]=EncoderValue[DataIndex]+in8(base + 8)*256; //read MSB
      SetOEBits(1,1);//OE2 High

      calcnewnums();
  }
```

This section of code is where the encoder is latched and read. It is only executed when all the bytes of the MicroStrain sensor have been received first; when ***SerialBit==10***. It is recorded after the last character is received because the time from the MicroStrain sensor position latch and the last character output is known. This means the encoder data can be shifted by a known amount to match encoder position to MicroStrain sensor position almost exactly in time. The code above is the recoding sequence of the 16-bit encoder, as discussed in the section on the encoder. ***SetOEBits*** gives the program a way to indicate which byte of data is needed and also when to latch the encoder position. OE1 is actually taken low then high twice in this code. It was discovered that the value outputted by the encoder is actually the value that was latched one latch previous. This means to get the position of the encoder at the exact time it was read the encoder has to be latched twice to get the correct data. If the encoder is not latched twice the data for the current position will not be outputted until the next read. The data outputted at the current read will be the data for the position at the previous read. The reason for this is unknown, however, it was proven that the data outputted was the previous data latch's position. The intermixed ***while*** statements poll the port B DIO looking for DA of the encoder to go low. As soon as this happens the data on the data lines from the encoder become valid and may be read. The time between lines of code seen in comment at line 29 is enough time to satisfy T2 and T3 of the encoder timing diagram. Further delay is not necessary for the encoder to function properly. The least significant byte of the position is read first, followed by the most significant byte. This is the reason the data read in line 32 is multiplied by 256 and then added to the least significant byte. The data is stored in the ***EncoderValue*** array. This array holds all the successive encoder readings. When the encoder is finished reading, the MicroStrain sensor data is then converted into useable numbers by ***calcnewnums()***. The code for this function is seen here:

```
void calcnewnums(void)
{
        dataid=SerialBuffer[0];
        roll=SerialBuffer[1]*256 + SerialBuffer[2];
        if(roll>=32768) roll=(-1)*(65536-roll);
        pitch=SerialBuffer[3]*256 + SerialBuffer[4];
        if(pitch>=32768) pitch=(-1)*(65536-pitch);
        yaw=SerialBuffer[5]*256 + SerialBuffer[6];
        if(yaw>=32768) yaw=(-1)*(65536-yaw);
        timerticks=SerialBuffer[7]*256 + SerialBuffer[8];
        checksum=SerialBuffer[9]*256+SerialBuffer[10];
}
```

This function combines and stores the current roll, pitch, and yaw Euler angles from the MicroStrain sensor in respective variables. The ***dataid*** should be the code for the data requested. In this case it is 14 or 0x0E. The roll, pitch, and yaw data most significant bytes and least significant bytes are combined into one number to represent each angle in a 16-bit form. The data outputted by the sensor is in two's compliment form. The ***if*** statement converts the numbers from a two's compliment into a signed integer form for ease of conversion to degrees later.

The interrupt service routine then continues by increasing the ***SerialBit*** variable by 1. This is used to track the number of characters taken off the buffer for the current data pack:

```
37     SerialBit++;
38     if(SerialBuffer[0]!=14) SerialBit=0; //reset because of
     invalid data
39     if((SerialBit==11)&&(DataIndex<DataSetSize))
40     {
41         SerialBit=0; //rollover for next data set
42         rollnums[DataIndex]=roll;
43         DataIndex++;
44     }
45
46     return 1;
47 }
```

The ***if*** statement in line 38 will reset the data pack if the first character received is not a 14, or 0x0E. It is expected that the first character of every new data packet will be this value because of the standard set by the MicroStrain sensor. The first set of the data packet sent will be command code for the data requested. In this project, the gyro-stabilized Euler angles are requested for which the command code is 0x0E. If the ISR places any character other than a 0x0E in the first index of the buffer array the index is reset. This will happen until 0x0E is found in the first index of the buffer. At this time, the

ISR knows that it is the start of a new valid data pack and will allow the index to increment.

Line 39 starts the routine if a complete data packet has been received. In this case the ***SerialBit*** should be 11, indicating 11 characters were read off the buffer, and the current data packet index is less than the size of the data set requested. If the ***while*** loop in the ***main*** function does not exit and the interrupt detached, it would be undesirable for data to continue to be recorded. When ***DataIndex*** is equal to or greater than ***DataSetSize***, the ISR knows that all the requested data has been processed and recorded; no further data is needed. Line 42 records the roll angle of the current reading into the array that will be output to the file later. Finally, the index of the data is incremented so when the ISR records the next set of data it will be placed in the next element of the two arrays. The ISR returns a 1, meaning the interrupt was processed correctly.

3. 10-Bit Encoder Program Differences

There were a few differences between the 16-bit encoder program described previously and the 10-bit encoder program. The main difference was the data collection method. The 10-bit encoder program used the polling method of MicroStrain data collection. This method is useful for a non-deterministic application, but for this application, the time between readings needs to be exact. The other difference was that the encoder was read when the request for data was sent out. This means the time difference between the returned data from the MicroStrain sensor and the encoder reading could be anywhere from 0-13ms of difference. The sensor would receive the request sometime during the current calculation cycle and then output that data. Since there is no way of knowing where in the calculation cycle the sensor is, there is no way of determining the time between the encoder reading and the sensor reading.

Another major difference in the two programs is the reading method of the encoder. Since the 10-bit encoder outputs data in a parallel method and there is minimal calculation time, the data from the encoder can be read all at once. This means the DIO of the PC104 reads all 10 bits at once. The 8 least significant bits are on port A and the 2 most significant bits are on port B. The data collected by the PC104 from the encoder will also have to be converted before output to the file. This is because the encoder outputs in grey code. This must be converted to a binary number and then converted to the final respective degree.

C. RESULTS

Many tests were performed. They included testing the 10-bit encoder at a pendulum length of two feet, 16-bit at one, two, and three foot lengths, and the pitch and yaw angles of the 16-bit encoder at a two foot length. The results were all very similar. There were definite characteristics present that were expected, but also some that were newly discovered. All of the results proved the 3DM-GX1 sensor does have the capability of operating within the specifications put forth by the MicroStrain company, under the speed conditions tested. A faster moving sensor may produce a larger error. Since the MicroStrain sensor and the encoder zero positions do not match, the test was started at the zero position for the first approximately 100 time samples and then the pendulum was put into motion. This period of no motion at the zero point allows the possibility to match the data up, based on this zero position. When the data is imported into MATLAB the mean of this rest period is taken for each data set. The mean is then subtracted from each data point to align the two sensors around zero degrees. The result of this is two sets of data that track the same degree position over time. After this, the error between the two sensors can be calculated. This procedure is performed for every plot of the data sets in this section.

The results from the 10-bit encoder can be seen in Figure 14.

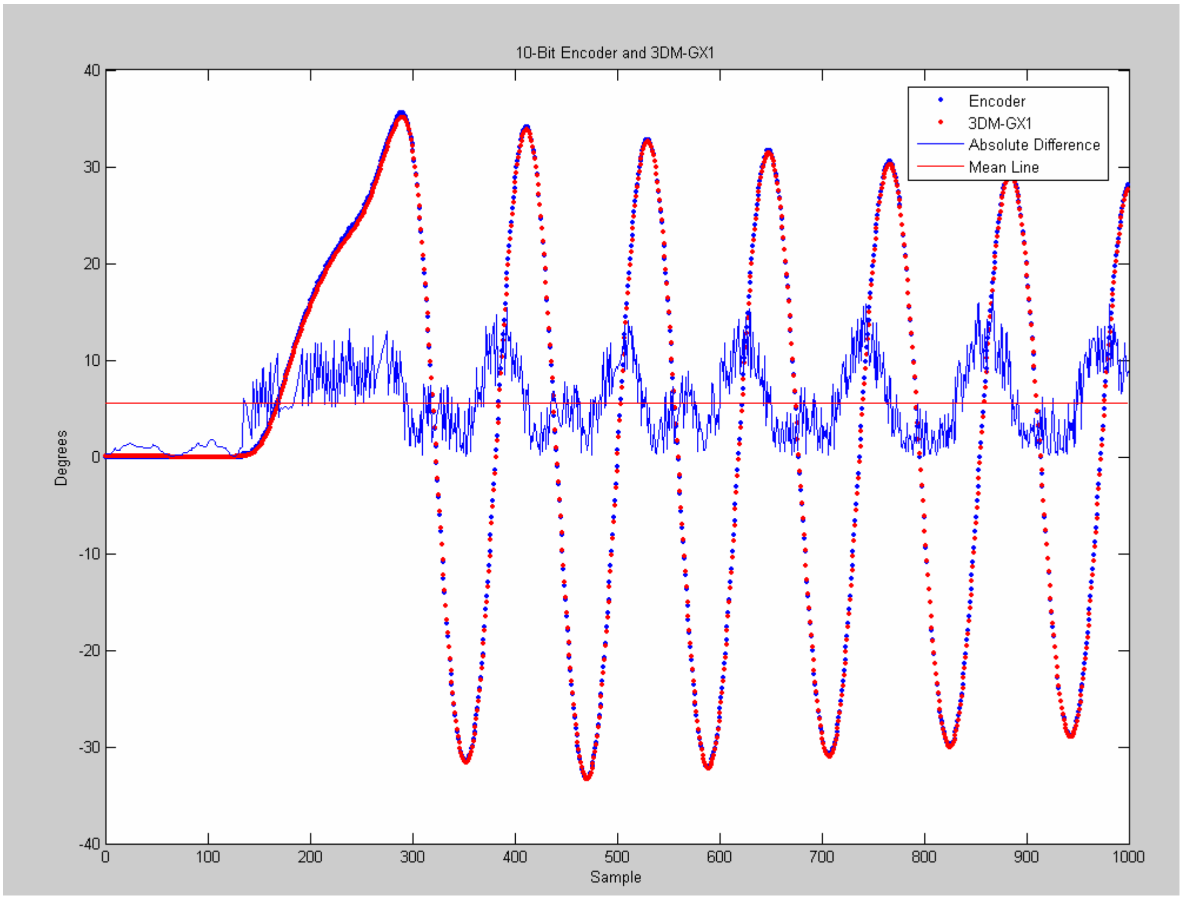


Figure 14. 10-Bit Encoder and 3DM-GX1 Sensor Results. The absolute difference is magnified by a gain of 20 in the plot.

This test was performed by allowing the pendulum to rest at the zero point for about 100 samples. The pendulum was then lifted and released by hand. The data collected with this sensor indicated that the largest error was only .8507°. This occurred at sample 867. The "Absolute Difference" is the absolute value of the difference between the readings of the two sensors for the same sample. The value is then multiplied by a gain of 20 so it would be easier to analyze on the same plot as the data. The mean line is the mean of this difference over the entire sample set. This line is there for reference purposes of the difference data. The MicroStrain sensor data in this example is shifted to the left by one sample. The method of data sampling used for this data collection does not ensure that the data matches at each sample period because the MicroStrain sensor reading is not guaranteed. Polling was used, so the reading associated with the time sample has the possibility of being one time sample ahead of the encoder, in position. This means at time “x”, the encoder was at the correct position, but the MicroStrain sensor reading at time “x” could be the reading that was supposed to be associated with time “x+1” of the encoder. This problem was solved with the 16-Bit encoder because the data collection was more deterministic and accurate. Another problem with the data collected by the 10-Bit encoder is the relation between the test bench, the encoder, and the MicroStrain sensor.

The test bench must have 16-bits of accuracy or better since the MicroStrain sensor has this degree of accuracy. The limited resolution of the 10-bit encoder is the cause of the stair characteristic of the zoomed view of the results plot in Figure 15. This was the motivation to move to the more accurate 16-bit absolute encoder.

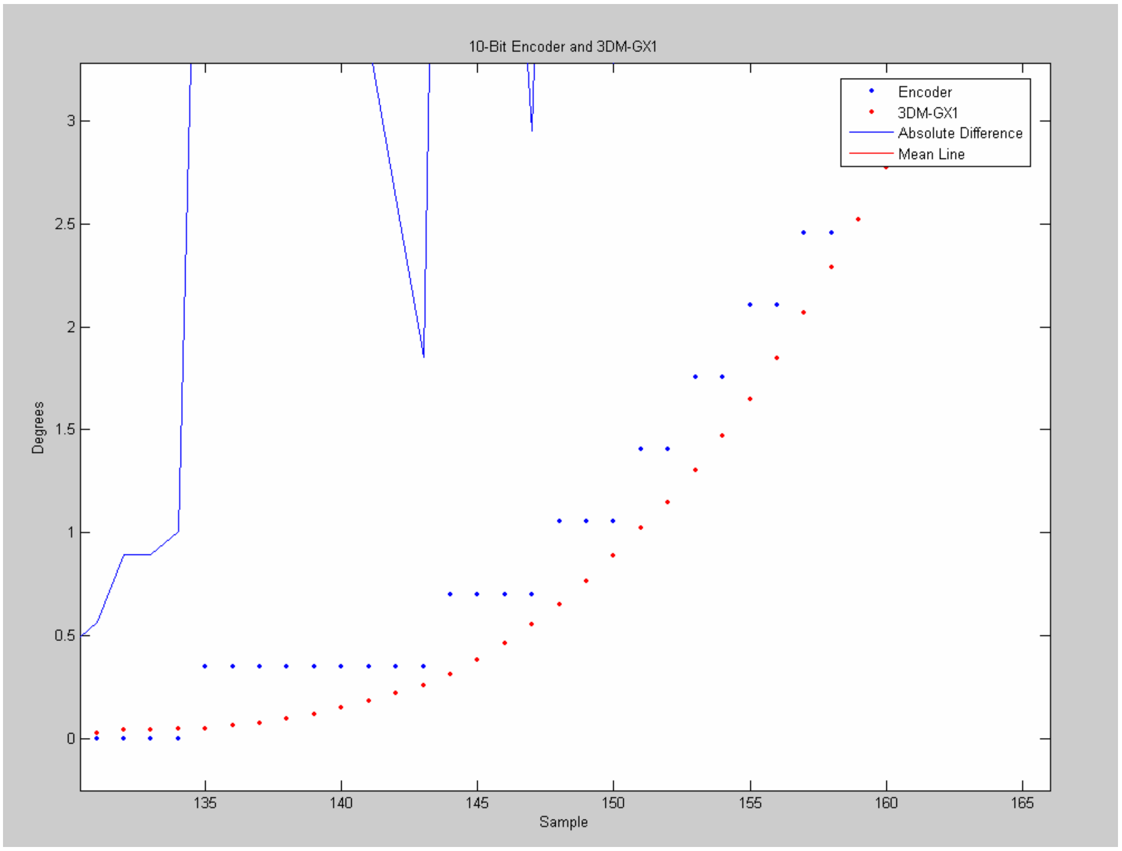


Figure 15. Zoomed View of 10-Bit Encoder and 3DM-GX1 Results.

The MicroStrain sensor was tested at three different lengths, one foot, two feet, and three feet. All of these results had very distinct characteristics that were more well defined in each test. All of the plots from this point on were generated in MATLAB using the same process. The major point that must be realized is the shift of the encoder data.

The data in the 10-bit encoder example was shifted, as it was for the 16-bit encoder, but the shift in the 16-bit encoder was much more accurate. The basic process of the MicroStrain sensor computation is to read all of the sensors, convert to a digital value, perform Euler calculations, and finally output on the RS-232 line. This process repeats continuously as long as the sensor is powered on and the command signal requests the Euler angles. The sensor deterministically starts a new calculation every 13.107ms. Therefore, it is known with relative assurance that at approximately the time the last character of the data is sent out of the sensor the next calculation is started. The sensor is read at the beginning of the cycle and the encoder is latched at this time. Thus, it can be said with relative assurance that the encoder position is 13.107ms later than the corresponding MicroStrain position, since This means the encoder position must be shifted back by 13.107ms to relate the positions at the exact time instead of the corresponding sensor sample. The position of the encoder does not have to be estimated at the time which is 13.107ms in the past, though, because it is known that the previous encoder position read was exactly 13.107ms in the past because the calculation cycle time of the MicroStrain sensor that is driving the data collection is 13.107ms. The un-shifted data can be found in Figure 16.

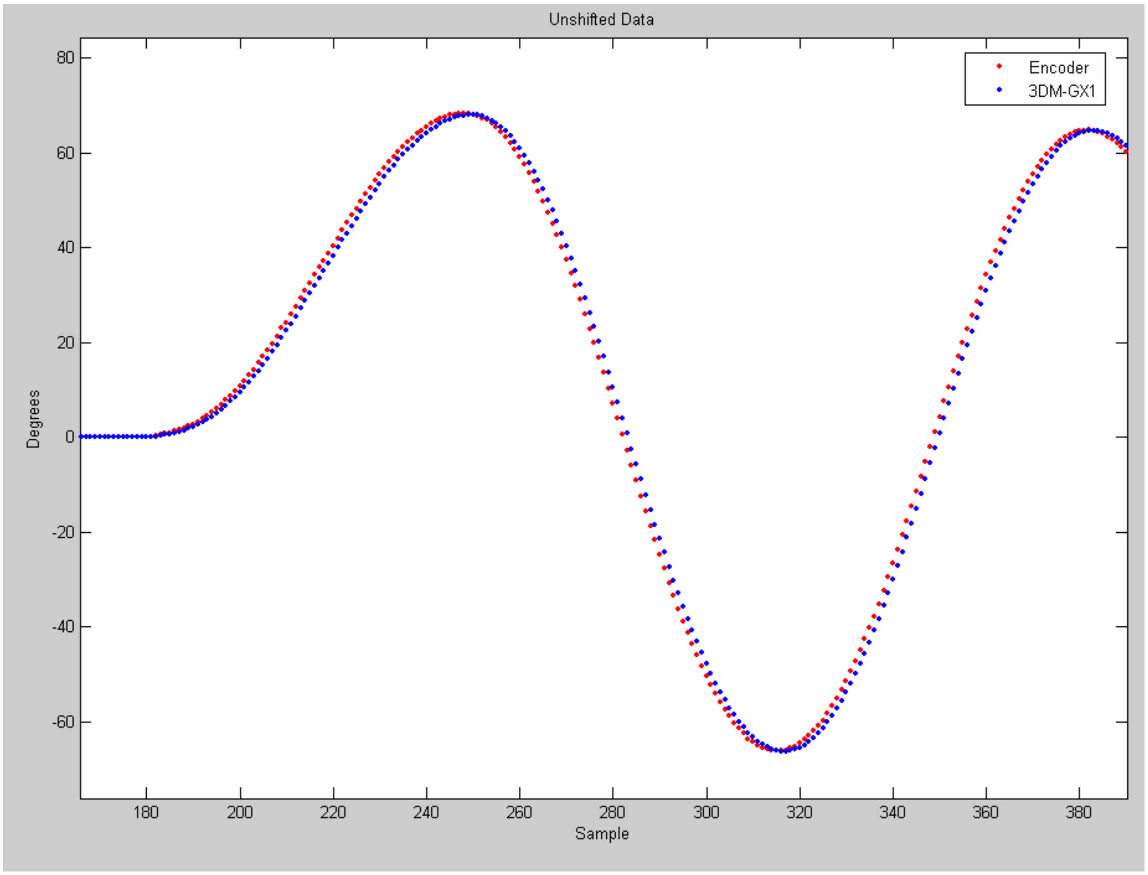


Figure 16. Un-shifted Encoder and MicroStrain Sensor Data.

The un-shifted data is the data that is received without the time correlation shift. It can be seen that the corresponding data has a gap of one time sample. The 3DM-GX1 data reaches a certain degree one time sample later than the encoder data. This time sample is the 13.107ms difference described earlier. To better analyze this data, the data must be shifted first.

In this project, instead of shifting the encoder position back 13.107ms in the past, the MicroStrain sensor data is shifted to correspond to the encoder data. This is accomplished by shifting the entire MicroStrain sensor data one index to the left, effectively matching the currently calculating sensor position with the encoder position at that same time. This erases the effect of the calculation cycle delay at the expense of loosing the first MicroStrain sensor data point read by the system. Once this data is matched up by time instead of sample, the data can be analyzed more accurately. This is exactly what was done in all of the following examples. The following plots are the one, two, and three feet test results.

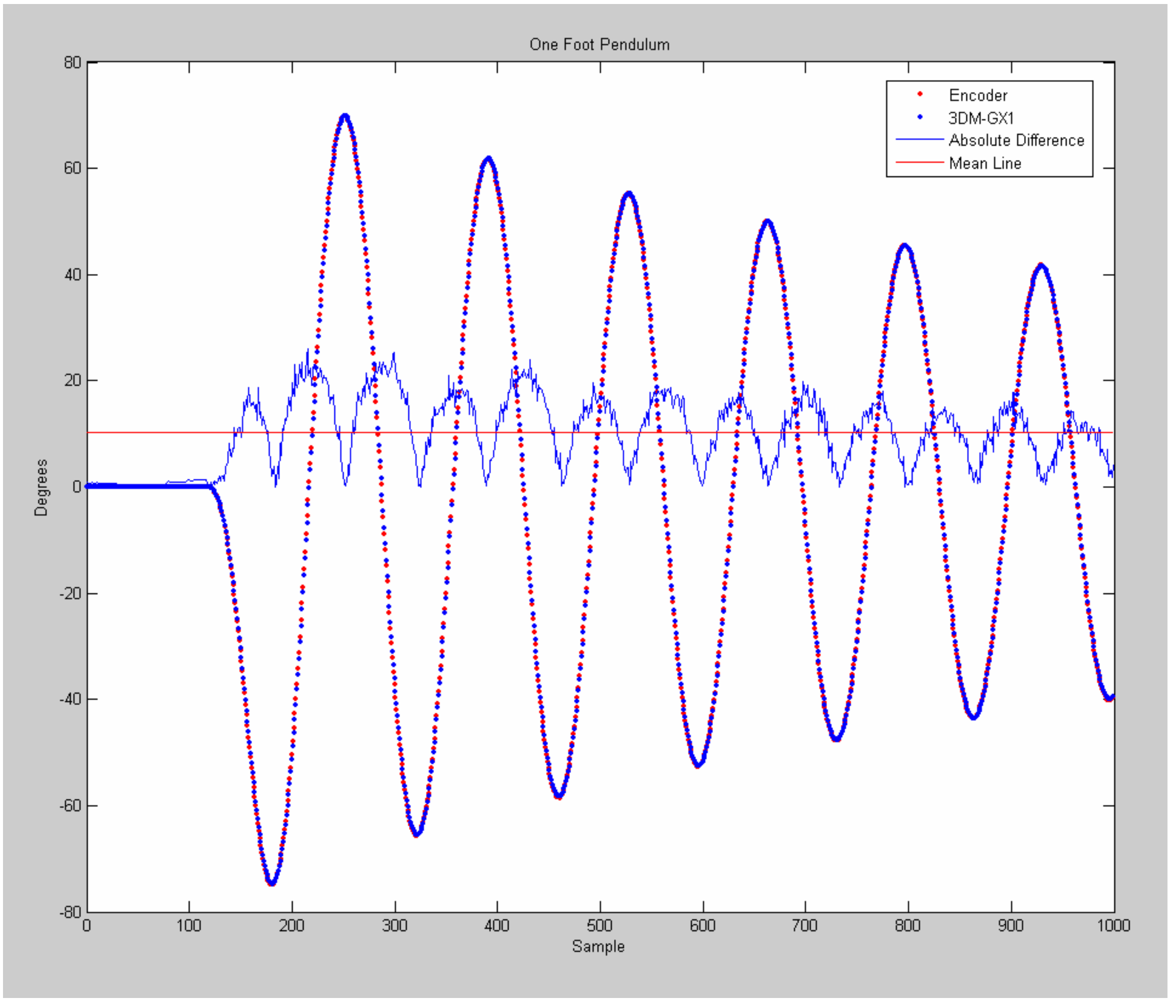


Figure 17. One Foot Pendulum Data. The absolute difference is multiplied by a gain of 20.

The data in the above plot is the data for the system when the pendulum is one foot in length. The mean line in this and all the following plots is the mean of the error. It was included to serve as a reference point for the error analysis. Accounting for the fact that the error is scaled by a gain of 20, is can be seen that the error is above 1° at various time points. The largest error is 1.3019° at sample 215. This error is within the specification of the MicroStrain sensor. The magnitude of the error also follows the expected pattern. It is expected that when the pendulum is moving the slowest there should be the least amount of error. This is because as the pendulum slows down, it approaches a static state. The static accuracy magnitude of the sensor is less than 0.5° of error. The position where the pendulum is the slowest is when it is changing direction at the extremes. Looking at the above plot, it can be seen that this is exactly what happens, as the pendulum approaches the maximum or minimum the error approaches 0.

Another curiosity observed is that the error is actually almost a direct function of the velocity. This can be observed in Figure 18.

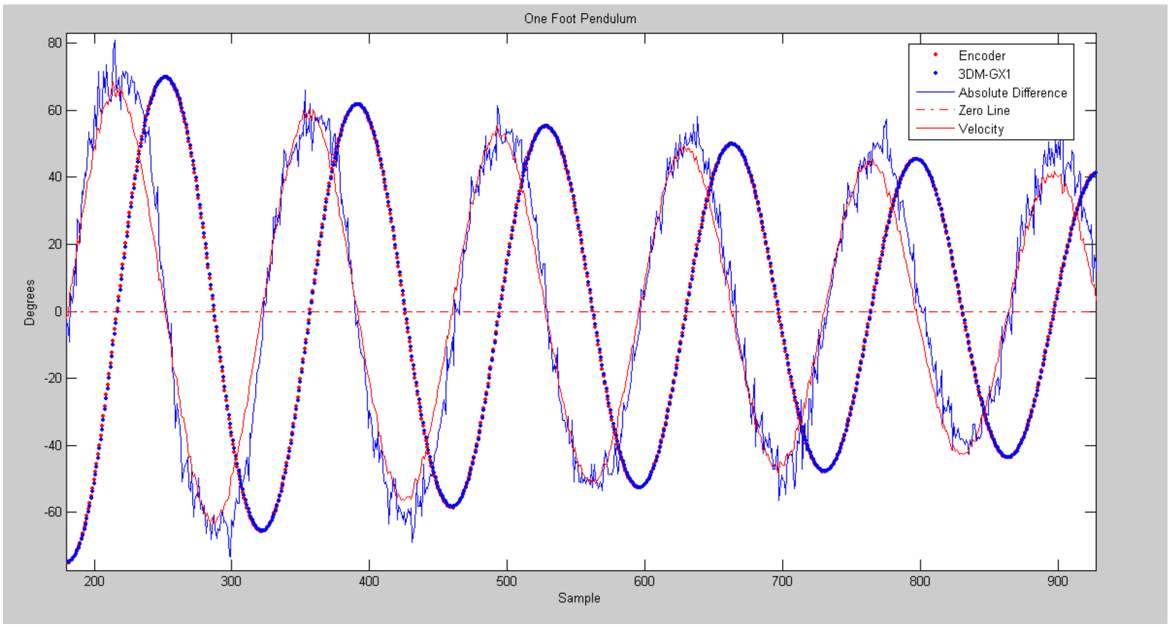


Figure 18. Zoomed View of One Foot Plot. The velocity has a gain of 20 and the absolute difference has a gain of 60.

In this plot, the absolute value of the error is not taken. The error data is the actual error, this time scaled by 60, not 20. This scale was used to more closely match the peaks of the curves. The velocity is the plot of the velocity of the encoder curve. This was found by taking the derivative of the encoder curve because velocity is the derivative of position. The derivative curve in the plot also has a gain of 20. This was used to match the tops of the curves so a visual analysis could be performed. In reality, the derivative curve is much smaller because the ***gradient*** function in MATLAB was used. This plot indicates that the equation for the error may not be the velocity exactly, but it is very similar. The exact reason for this error is not known.

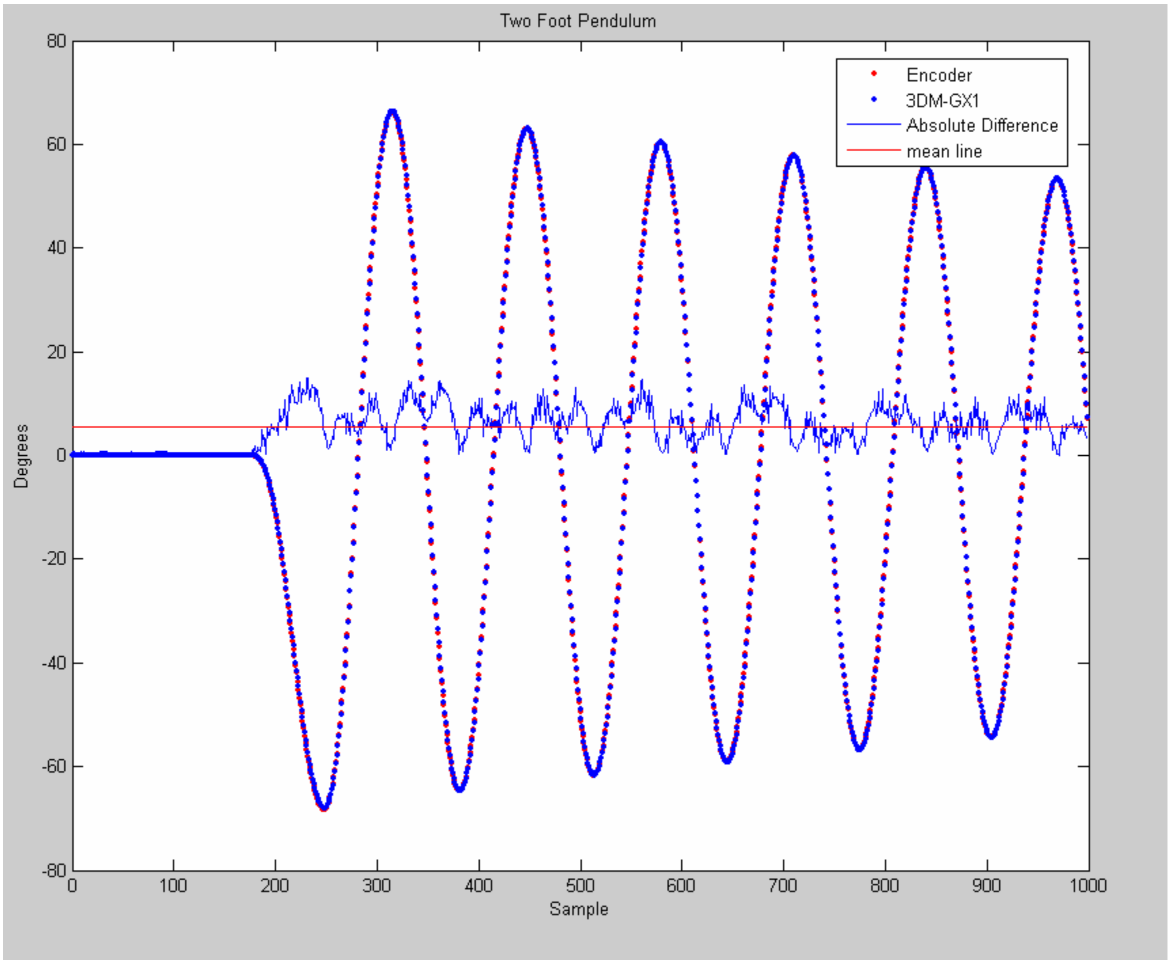


Figure 19. Two Foot Pendulum Data. The absolute difference is multiplied by a gain of 20.

The two foot plot is very similar to the one foot plot. This two foot plot can be found in Figure 19. The recorded max error is much better though. The max error is .7459° at sample 561. This is again within the specifications for the MicroStrain sensor. This is most likely because the pendulum is moving slower. As the pendulum length increases, the speed of oscillation decreases. Holding the correlation discovered for the previous test, this is what was expected; the slower the velocity, the more accurate the readings. There is one slight difference though. There is a small camel back at the peaks of all the error curves. A more pronounced view is seen in Figure 20.

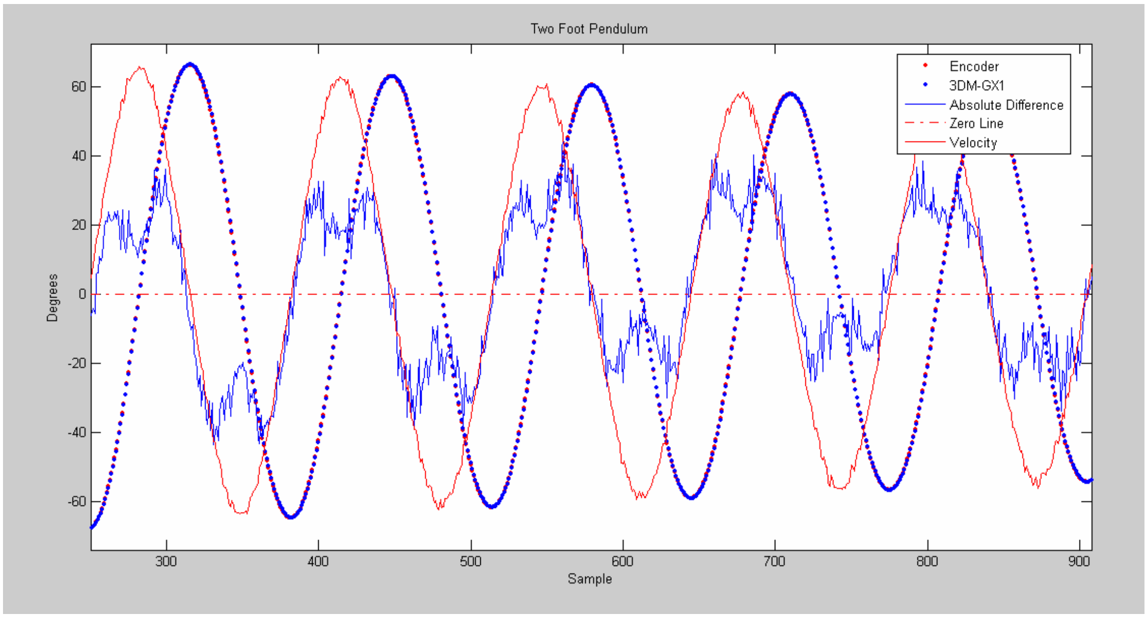


Figure 20. Zoomed View of Two Foot Plot. The velocity has a gain of 20 and the absolute difference has a gain of 60.

The camel back could be caused by two possible effects. It is possible that the acceleration has a small part in the equation for the error. This would account for the camel back because at the camel backs, the acceleration is zero. The acceleration would have to be inversely related to the velocity. That is, when the velocity magnitude is greater, the acceleration would have less of an effect. This would account for the differences between this plot and the plot of the one foot test. In the one foot test the velocity is greater, meaning the acceleration has less of an impact.

This would allow the one foot curve to be smooth and the two foot curve to have a little camel back when the acceleration is zero.

The second cause of the camel back could be due to the flexure of the pendulum shaft. The shaft is made from wood and has a small spring characteristic in it. As the pendulum swings the shaft may flex slightly. The longer the pendulum, the larger the flex. The flex would also be periodic because of the spring like characteristic of wood. When the wood is flexed in one direction it will reverse the flex in the other direction when the force holding the flex is overcome. This motion would create a periodic motion, which is exactly what the camel back is. If this is the cause of the camel back, the camel back found here should be more pronounced in the three foot test, because the pendulum is longer and has more ability to flex.

The three foot test results can be seen in Figure 21.

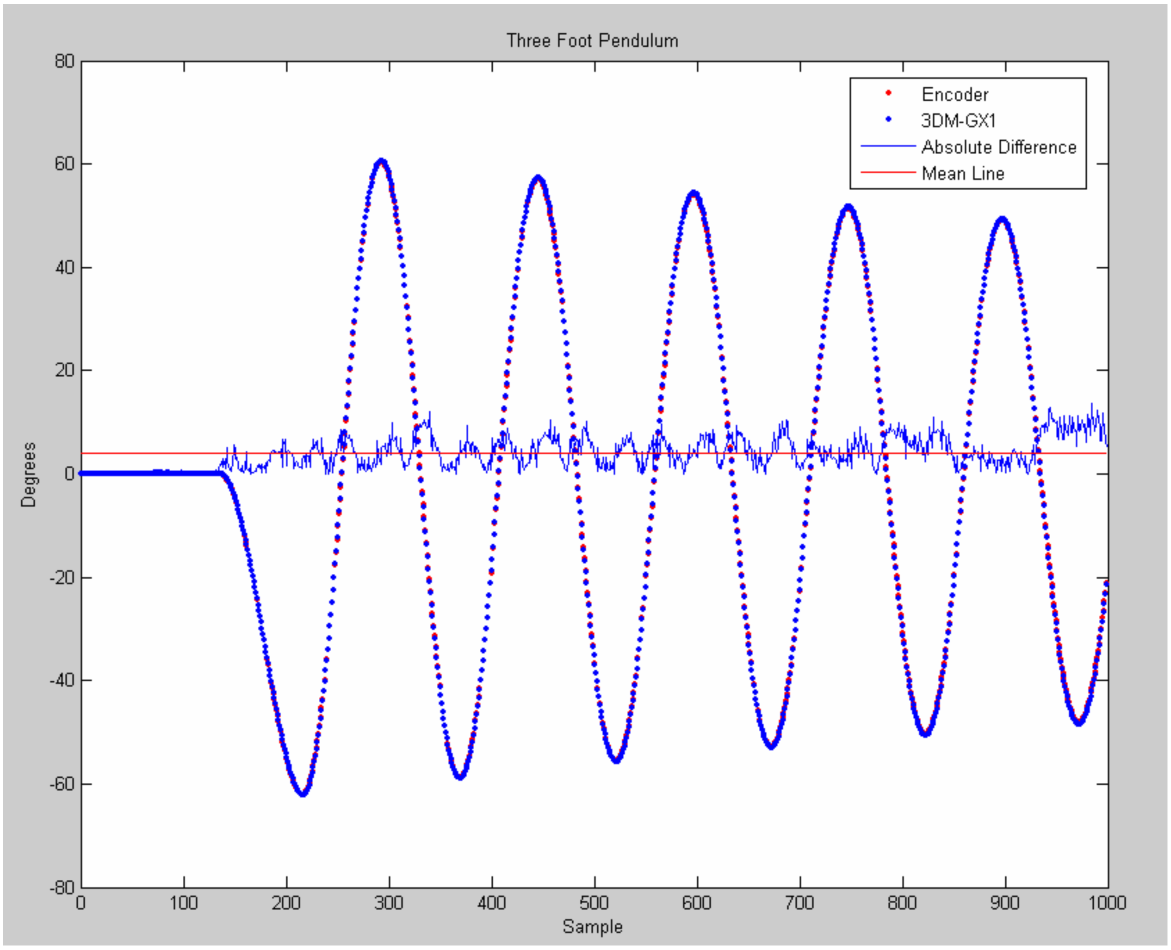


Figure 21. Three Foot Pendulum Data. The absolute difference is multiplied by a gain of 20.

The maximum error for the three foot test is .6820° at sample 984. Once again, as can be expected, the sensor error is well within the MicroStrain specifications. The plot clearly shows the MicroStrain sensor’s position very closely matches the encoder position for the entire path of motion. There is some peculiarity in this plot as well though. In this plot the error data does not seem to have much correlation to the position at all. It certainly does not have any correlation to the velocity, as seen in Figure 22.

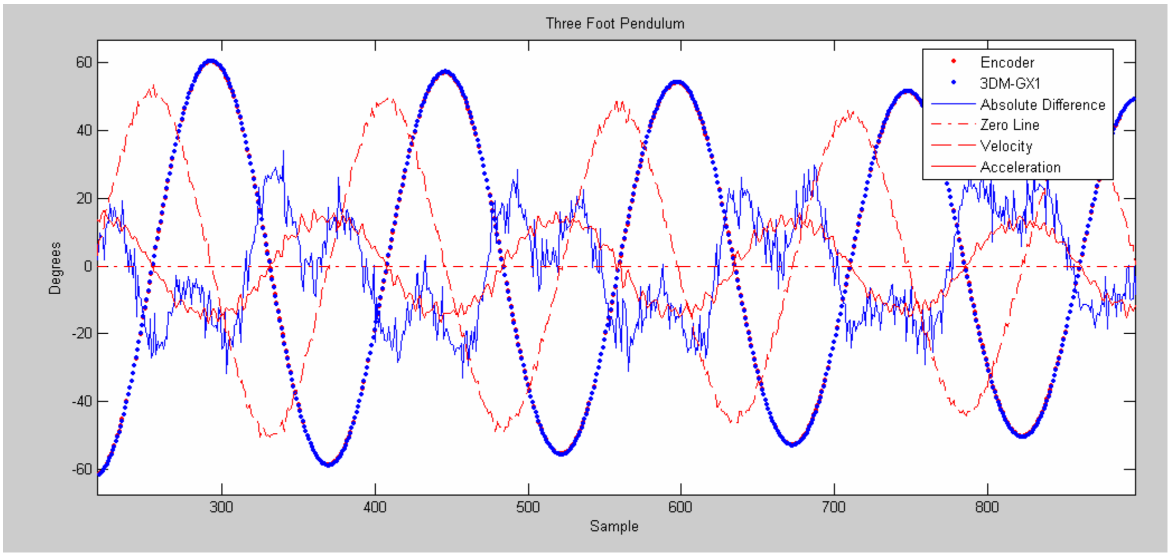


Figure 22. Zoomed View of Three Foot Plot. The velocity and the acceleration both have a gain of 20 and the absolute difference has a gain of 60.

On the above plot, the acceleration was plotted. This is the second derivative of position and in this plot the second derivative of the encoder data. A gain was applied to the curve for visual analysis. A mean filter was also applied to the curve to make it smoother. The filter took the average of a point and the one immediate point on each side of it to create a new point. This process was applied to every point which resulted in a much smoother curve. It is very peculiar that error curve for this length is very similar to the acceleration curve of the pendulum. This is what was expected though, based on the theories developed with the one and two foot tests. The differences between the acceleration curve and the error curve are relatively small, suggesting a relationship. The differences could be due to the flexure of the rod as discusser earlier.

The overall results indicate that the error of the MicroStrain sensor is very likely a function of the velocity, acceleration, and flexure of the pendulum rod. The faster the pendulum moves the more influence the velocity has. As the movement of the pendulum slows, the acceleration starts to have more of an influence on the error. The longer the pendulum, the more flexure and associated error influence. The error from the one foot test to the three foot test seems to have a logarithmic property. This could be caused by the changes in factors influencing the error. This may also be a clue to the relationship of acceleration, velocity, and flexure to the error of the MicroStrain sensor. It also indicates the possibility of the MicroStrain sensor error eventually becoming larger than the specifications indicate. The exact point where this may occur is not known because there is no formula for the error relationship, only clues that could eventually lead to one. It is also possible that the error will remain within specifications until the sensor's motion moves outside the maximum angular rate specification of the sensor. At this point it is expected that the error and other measurements will not be accurate.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The purpose of this project was to develop a control systems environment complete with encoder capabilities on a PC104. The system capabilities were tested by using it to measure the dynamic accuracy of the MicroStrain 3DM-GX1 orientation sensor against the benchmark of an encoder. The digital input/output and the digital-to-analog conversion circuits and the RS-232 port of the PC104 were used to accomplish this task.

In this project, a Diamond Systems Prometheus PC104 with a data acquisition circuit was used as a target machine. The main encoder used was a 16-bit absolute encoder by Gurley Precision Instruments. The encoder communication and control was performed with the digital input/output circuit. The MicroStrain 3DM-GX1 sensor communications all occurred through the RS-232 port. The MicroStrain sensor was attached to the end of a wooden pendulum and the encoder served as the pivot point of this pendulum. This allowed the angles of the two sensors to move synchronously with one another. The pendulum was set in motion and the position of the two sensors were recorded and outputted to a MATLAB compatible file.

The results yielded an error of the MicroStrain sensor well within the specifications established by the company. The worst error was 1.3019° when the pendulum was at a length of one foot. It was discovered that the error is a function of the velocity, acceleration, and flexure of the pendulum.

B. RECOMMENDATIONS

Further work would be deriving an error equation for the MicroStrain sensor. This would be useful in any application that uses this sensor because it may allow the user to compensate for the error, virtually erasing it. An encoder board could also be added to the PC104 system to allow interaction with the incremental encoder and multiple encoders at once. The system also can be used to test the dynamic accuracy of other inertia-based orientation sensors.

APPENDIX A: 16-BIT ENCODER C++ CODE

```cpp
#include <control/control.wse>

#define sbase 0x3F8
int SerialBit,DataIndex,DataSetSize;
int roll,pitch,yaw,dataid,timerticks,checksum;
void SerComInit(void);
void calcnewnums(void);
volatile signed rollnums[100000];
volatile signed SerialBuffer[11];
volatile signed EncoderValue[100000];
void SendInstantEulerAngleCONT(void);
void StopCONTmode(void);
void FileOutput(void);
void SetOEBits(int,int);

int isr4_routine()
{
    if(SerialBit<11) SerialBuffer[SerialBit]=in8(sbase + 0); //read a character off buffer
    if(SerialBit==10) //Get all Encoder Data and calc roll pitch and yaw
    {
        SetOEBits(0,1);//OE1 Low
        while(in8(base + 9) & 1){}
        SetOEBits(1,1);//OE1 High
        //1.8us
        SetOEBits(0,1);//OE1 Low
        while(in8(base + 9) & 1){}
        EncoderValue[DataIndex]=in8(base + 8); //read LSB
        SetOEBits(1,1);//OE1 High
        //1.6us
        SetOEBits(1,0);//OE2 Low
        while(in8(base + 9) & 1){}
        EncoderValue[DataIndex]=EncoderValue[DataIndex]+in8(base + 8)*256; //read MSB
        SetOEBits(1,1);//OE2 High

        calcnewnums();
    }
    SerialBit++;
    if(SerialBuffer[0]!=14) SerialBit=0; //reset because of invalid data
    if((SerialBit==11)&&(DataIndex<DataSetSize))
    {
        SerialBit=0; //rollover for next data set
        rollnums[DataIndex]=roll;
        DataIndex++;
    }

    return 0;
```

```
}

int main()
{
    BoardInit()

    out8(base + 11, 0xFE); //set all DIOA DIOB to input DIOClow
  Output
    out8(base+10,0x03);//OE1 OE2 high

    SerialBit=0;
    DataIndex=0;
    DataSetSize=1000;

    SerComInit();
    StopCONTmode();
    ISR4Init();

    SendInstantEulerAngleCONT();

    while(DataIndex<DataSetSize)
    {
        if(checksum==dataid+roll+pitch+yaw+timerticks)
        {
            cout<<"Its good data"<<endl;
        }
        else cout<<"BAD DATA!!!!"<<checksum<<"
  "<<dataid+roll+pitch+yaw+timerticks<<endl;
    }

    //stop continuous mode
    StopCONTmode();
    InterruptDetach(isr4handid);
    FileOutput();

    cout<<"Done";
    return 1;
}

int isr_routine(void)
{
    return 0;
}

/*Initialize COM with
 * 384000 baud
 * 1,N,8 */
void SerComInit(void)
{
    out8(sbase + 4, 0x09); //MCR data terminal ready
    out8(sbase + 3, 0x83); //enable latch for baud rate set
    out8(sbase + 0, 0x03); //lower baud divisor
    out8(sbase + 1, 0x00); //upper baud divisor
    out8(sbase + 3, 0x03); //disable latch, 1,N,8
    out8(sbase + 2, 0x07); //set FIOF to 1 character
    out8(sbase + 1, 0x15); //enable recv interupt
```

```
}

void calcnewnums(void)
{
        dataid=SerialBuffer[0];
        roll=SerialBuffer[1]*256 + SerialBuffer[2];
        if(roll>=32768) roll=(-1)*(65536-roll);
        pitch=SerialBuffer[3]*256 + SerialBuffer[4];
        if(pitch>=32768) pitch=(-1)*(65536-pitch);
        yaw=SerialBuffer[5]*256 + SerialBuffer[6];
        if(yaw>=32768) yaw=(-1)*(65536-yaw);
        timerticks=SerialBuffer[7]*256 + SerialBuffer[8];
        checksum=SerialBuffer[9]*256+SerialBuffer[10];
}

void SendInstantEulerAngleCONT(void)
{
    out8(sbase + 0, 0x10); //Command command
    out8(sbase + 0, 0x00); //Null command
    out8(sbase + 0, 0x0E); //Send GYRO Euler Angles command
}

void StopCONTmode(void)
{
    out8(sbase + 0, 0x10); //Command Command
    out8(sbase + 0, 0x00); //Null Command
    out8(sbase + 0, 0x00); //Null Command
}

void FileOutput(void)
{
    float myoutf;
    FILE* myfile;

    myfile=fopen("NewEncoder23.m","w+");
    fprintf(myfile,"Encoder=[");
    for(int i=0;i<DataSetSize;i++)
    {
        fprintf(myfile," %d\n",EncoderValue[i]);
    }
    fprintf(myfile,"]';");

    fprintf(myfile,"\n\nMicroStrain=[");
    for(int i=0;i<DataSetSize;i++)
    {
        myoutf=((float)rollnums[i])*360.0/65536.0;
        fprintf(myfile," %f\n",myoutf);
    }
    fprintf(myfile,"]';");
}

void SetOEBits(int OE1, int OE2)
{
    out8(base+10,(OE1 + 2*OE2));
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: CONTROL.WSE CODE

```
#include <cstdlib>
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
#include <iomanip.h>
#include <time.h>
#include <unistd.h>
#include <hw/inout.h>
#include <sys/neutrino.h>
#include <math.h>

#define base 0x280

volatile unsigned nums[100000];
volatile unsigned isrhandid;
int frequency;

int BoardInit(void);
int waitforADconversion(void);
int waitforADsettle(void);
int DAcheckstatus(void);
int SendDAC(int , float);
float GetADVolts(int, int);
int fileoutput(const char* , int);
int Timer0Init(int , int, int);
int Timer0Start();
int SetFIFOThreshold(int);
int ADInit(int,int,int,int,int,int);
int ClearAnalogInterrupt(void);
int ResetFIFOdepth(void);
int isr_routine(void);
int Binary2Dec(int,int);
int* Grey2Binary(int,int);
int* DIO2Grey(int,int);
int PortAin(void);
int PortBin(void);
void SetAllDIO(int);
const struct sigevent* isr4_handler(void*, int);
int isr4_routine(void);
int int4;
volatile unsigned isr4handid;
int ISR4Init(void);

// This must be run to get root permission at access hardware
// It also resets the data aquisition board, everything except the
   DAC output
int BoardInit(void)
{
    int privity_err;
     privity_err=ThreadCtl( _NTO_TCTL_IO, NULL ); // thread gets
   root permission at access hardware
```

```
    if(privity_err==-1)
    {
        cout<<"can't get root permission";
        return -1;
    }
    out8(base + 0, 0x40); //reset the board, except the DAC output
    return 1;
}

// This function waits for the AD conversion to happen
// Returns 0 if successful
// Reutrns -1 if timed out
int waitforADconversion()
{
    int i;
    for(i=0;i<10000;i++)
    {
        if(!(in8(base + 3) & 0x80)) return 0;
    }
    return -1;
}

// This function waits for the AD circuit to settle out\
// Returns 0 if successful
// Returns -1 if timed out
int waitforADsettle()
{
    int i;
    for(i=0;i<10000;i++)
    {
        if(!(in8(base + 3) & 0x20)) return 0;
    }
    return -1;
}

// This loops until the DA conversion is completed
// Returns 0 if successful
// Returns -1 if timed out
int DAcheckstatus()
{
    int i;
    for(i=0;i<10000;i++)
    {
        if(!(in8(base + 3) & 0x10)) return 0;
    }
    return -1;
}

// Output a voltage volts on channel chan
int SendDAC(int chan, float volts)
{
    int DAlevel;
    int DAref=10;
    if(volts<-10) volts=-10;
    if(volts>10) volts=10;
    DAlevel=chan*16384+(volts*2048)/DAref+2048;
    out16(base + 6, DAlevel);
    DAcheckstatus();
    return 1;
}

float GetADVolts(int chan, int gain)
{
    int ADvalue;
    float voltage;
    out8(base + 2, chan*16 + chan); //set the channel range
    out8(base + 3, 4*1 + gain); //set scan mode and gain
    waitforADsettle();
    out8(base + 0, 0x80); //start AD conversion
    waitforADconversion();
    ADvalue=in8(base + 0) + in8(base + 1) * 256;
    if(ADvalue<32768) voltage= .0000152587890625 * 20 * ADvalue;
    else voltage=.0000152587890625*20 * ADvalue - 20;
    return voltage;
}

int fileoutput(const char* filename, int numsamples)
{
    FILE* myfile;
    float voltage;
    myfile=fopen(filename,"w+");
    fprintf(myfile,"Hello=[");
    for(int i=0;i<numsamples;i++)
    {
          if(nums[i]<32768)  voltage=  .0000152587890625  *  20  *
   nums[i];
    else voltage=.0000152587890625*20 * nums[i] - 20;
        fprintf(myfile," %f\n",voltage);
    }
    fprintf(myfile,"]';");
    return 0;
}

//This  function  initializes  Timer0  with  a  corrected  frequency,
   rounding
//up to the next highest divisor of the input clock
//freq is desired frequency, input is clock source, gating will
   enable the gating
//input: 0=10Mhz 1=1Mhz;
int Timer0Init(int freq, int input, int gating)
{
    long working1,working2,entry; //workingvalues
    int hexout[3];
    int regstat;
    entry=10000000/freq;
    frequency=10000000/entry;
    for(int i=0;i<3;i++)
    {
            working1=entry/pow(16.0,2.0*i);
            working2=entry/pow(16.0,2.0*i+2);
            hexout[i]=working1-256*working2;
    }
    regstat=(in8(base + 4) & 0x20) / 0x20;//check clock input
    if(regstat!=input) //if clock input is not what is desired
    {
        if(input==1) out8(base + 4, (in8(base + 4)) + 0x20); //set it to 1Mhz
        else out8(base + 4, (in8(base + 4)) - 0x20); //set it to 10Mhz
    }

    out8(base + 12, hexout[0]); //low value
    out8(base + 13, hexout[1]);//mid value
    out8(base + 14, hexout[2]);//high value
    out8(base + 15, 0x02);//load value
    //out8(base + 15, 0x04);//start counting
 }

 //start Counter0, Timer0Init must be run first
 int Timer0Start(void)
 {
    out8(base + 15, 0x04);
 }

//sets the threshold of the FIFO
int SetFIFOThreshold(int depth)
{
    out8(base + 5, depth);
}

 int ADInit(int lowchan, int highchan, int scan, int interrupt, int trigger, int gain)
{
    int regstat,input;
     if(lowchan>highchan) cout<<"Error: The AD high channel is lower than the low channel"<<endl;
    else if((lowchan<0) || (lowchan>15)) cout<<"Error: The AD low channel is either less than 0 or greater than 15"<<endl;
     else if((highchan<0) || (highchan>15)) cout<<"Error: The AD high channel is either less than 0 or greater than 15"<<endl;
    else if((scan<0)||(scan>1)) cout<<"Error: The AD scan value is invalid"<<endl;
     else if((interrupt<0)||(interrupt>1)) cout<<"Error: The AD interrupt value is invalid"<<endl;
    else if((trigger<0)||(trigger>1)) cout<<"Error: The AD trigger value is invalid"<<endl;
        else if(!((gain==1)||(gain==2)||(gain==4)||(gain==8))) cout<<"Error: The AD gain value is invalid"<<endl;
    else
    {
         out8(base + 2, lowchan+16*highchan); //set the channel range
        out8(base + 3, 4*scan+logf(gain)/log(2.0)); //set the gain and scan mode
        regstat=(in8(base + 4) & 0x10) / 0x10; //check AD trigger source
        if(regstat!=trigger) //if trigger source is not what is desired
        {
            if(trigger==1) out8(base + 4, (in8(base + 4)) + 0x10); //set it to external clock input
            else out8(base + 4, (in8(base + 4)) - 0x10); //set it to Counter0
        }
        regstat=(in8(base + 4) & 0x01) / 0x01; //check interrupt enable
        if(regstat!=interrupt) //if interrupt enable is not what is desired
        {
            if(interrupt==1) out8(base + 4, (in8(base + 4)) + 0x01); //enable interrupt
            else out8(base + 4, (in8(base + 4)) - 0x01); //disable interrupt
        }
        waitforADsettle(); //let the AD converter to settle out
        return 1;
    }
    cout<<"ADInit Error, program many not execute as expected"<<endl;
    return 0;
}


int ClearAnalogInterrupt(void)
{
    out8(base + 0, 0x01);
    return 1;
}

int ResetFIFOdepth(void)
{
    out8(base + 0, 0x10);
    return 1;
}

int ADvalue(void)
{
    return (in16(base + 0));
}

int NormADvalue(void)
{
    int AD;
    AD=in16(base + 0);
    if(AD<32768) return AD;
    else return (65535-AD)*(-1);
}

int isrSendDAC(int chan, int output)
```

```
{
    output=output+2048;
    if(output>4095) output=4095;
    if(output<0) output=0;
    out16(base + 6, (chan*16384+output));
    DAcheckstatus();
    return 1;
}

/* The hardware interrupt handler */
const struct sigevent* isr_handler(void *arg, int intr)
  {
    isr_routine();
    InterruptUnmask(5,isrhandid);
  }

int ISRInit(void)
{
            isrhandid=InterruptAttach(5,isr_handler,   NULL,   0,
   _NTO_INTR_FLAGS_TRK_MSK);
        return 1;
}

int ISRStop(void)
{
    InterruptDetach(isrhandid);
    return 1;
}

int CheckFIFOOVF(void)
{
    return ((in8(base+3)&0x08)/0x08);
}

/*Convert a binary number into decimal*/
int Binary2Dec(int* Binary,int size)
{
    int Decimal;
    Decimal=0;
    for(int i=0;i<size;i++)
    {
        //the Binary array member must be referenced by
        //*(Binary+i)
        Decimal=Decimal+(*(Binary+i))*(int)(pow(2.0,i));
    }
    return Decimal;
}

/*Convert a grey code number into binary*/
int* Grey2Binary(int* Grey,int size)
{
     int*  Binary  =  new  int[size];  //this  is  needed  for  proper
   return
    //Fill Binary with all 1's
    for(int i=0;i<size;i++)
    {
        Binary[i]=1;
    }
    //Start Conversion
    Binary[size-1]=*(Grey+(size-1));
    for(int i=size-1;i>0;i--)
    {
        //Grey array must be referenced by *(Grey+i)
        if(Binary[i]==(*(Grey+(i-1))))   Binary[i-1]=0;
    }
    return Binary; //return binary number
}

/*convert the DIO raw number into grey code*/
int* DIO2Grey(int DIO, int size)
{
        int* Grey = new int[size]; //this is needed for the proper
   return for array

        //Strip off bits through masking
        Grey[0]=((DIO & 1)==1);
        Grey[1]=((DIO & 2)==2);
        Grey[2]=((DIO & 4)==4);
        Grey[3]=((DIO & 8)==8);
        Grey[4]=((DIO & 16)==16);
        Grey[5]=((DIO & 32)==32);
        Grey[6]=((DIO & 64)==64);
        Grey[7]=((DIO & 128)==128);
        Grey[8]=((DIO & 256)==256);
        Grey[9]=((DIO & 512)==512);

        return  Grey; //return the grey code
}

/*Reads PortA*/
int PortAin(void)
{
    int read;
    read=in8(base + 8); //read port A
    return read;
}

/*Reads PortB*/
int PortBin(void)
{
    int read;
    read= in8(base + 9); //read port B
    return read;
}

/*Set all the DIO ports to input/output, 1=in 0=out*/
void SetAllDIO(int inout)
{
    if(inout==1) out8(base + 11, 0xFF); //make all DIO ports input
    else out8(base + 11, 0x00); //Make all DIO ports output
```

```
}

/*IRQ 4 Handler*/
const struct sigevent* isr4_handler(void *arg, int intr)
{
    isr4_routine(); //ISR routine
    InterruptUnmask(4,isr4handid); //Enable IRQ 4
}

/* Initialize IRQ 4 handler*/
int ISR4Init(void)
{
        isr4handid=InterruptAttach(4,isr4_handler,     NULL,     0,
   _NTO_INTR_FLAGS_TRK_MSK);
}
```

LIST OF REFERENCES

[1] X. Yun and E.R. Bachmann, "Design, implementation, and experimental results of a quaternion-based Kalman filter for human body motion tracking," ***IEEE Transactions on Robotics***, Vol. 22, No. 6, pp. 1216-1227, December 2006,

[2] D. Churchill, "Quantification of human knee kinematics using the 3DM-GX1 sensor," MicroStrain Inc., January 2004.

[3] "Ampro History," May 2007, http://www.ampro.com/html/Overview_History.html.

[4] PC104 Consortium, "History," May 2007, http://www.pc104.org/history.html.

[5] ***Prometheus High Integration PC/104 CPU with Ethernet and Data Acquisition Models PR-Z32-E-ST PR-Z32-EA-ST User Manual V1.44,*** Diamond Systems Corporation, 2003.

[6] QNX Neutrino Realtime OS: Kernel Benchmark Mehtodology, QNX Software Systems Ltd., 2003.

[7] S. Furr, "What is real time and why do I need it?" QNX Software Systems Ltd., 2002.

[8] S. Arms, "MicroStrain message from the president," May 07, http://www.microstrain.com/company-overview.aspx.

[9] "3DM-GX1," May 2007, http://www.microstrain.com/3dm-gx1_specs.aspx.

[10] ***3DM-GX1 Data Communications Protocol***, MicroStrain Inc., 2005.

[11] ***3DM-GX1 Orientation Sensor Timer Ticks, Calculation Cycle and Data Output Rates,*** MicroStrain Inc., 2006.

[12] ***CP-300 Series Housed Encoders,*** Computer Optical Products Inc.

[13] ***Optical Encoder Applications,*** Computer Optical Products Inc.

[14] *Gurley Model A58 Absolute Encoder,* Gurley Precision Instruments.

[15] "Serial UART information," May 2007, http://www.lammertbies.nl/comm/info/serial-uart.html.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Jeffery Knorr, ECE Department Chairman
   Naval Postgraduate School
   Monterey, California

4. Xiaoping Yun
   Naval Postgraduate School
   Monterey, California

5. Matthew Feemster
   United States Naval Academy
   Annapolis, Maryland

6. Douglas Fouts
   Naval Postgraduate School
   Monterey, California

7. James Calusdian
   Naval Postgraduate School
   Monterey, California

8. Jonathan Shaver
   Naval Postgraduate School
   Monterey, California